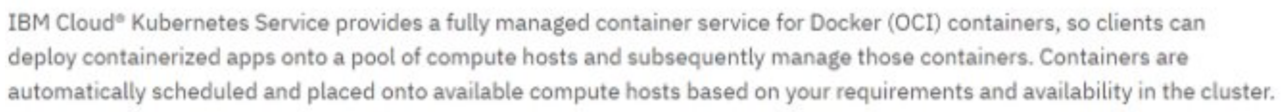
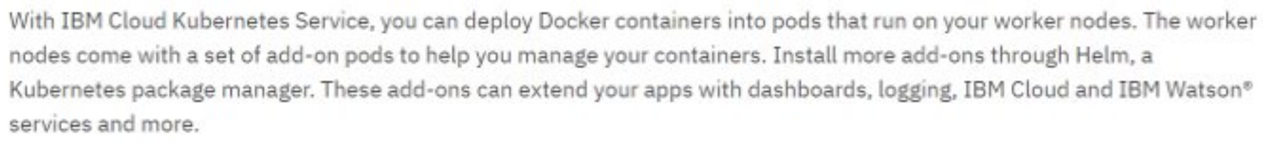
Exhibit 4

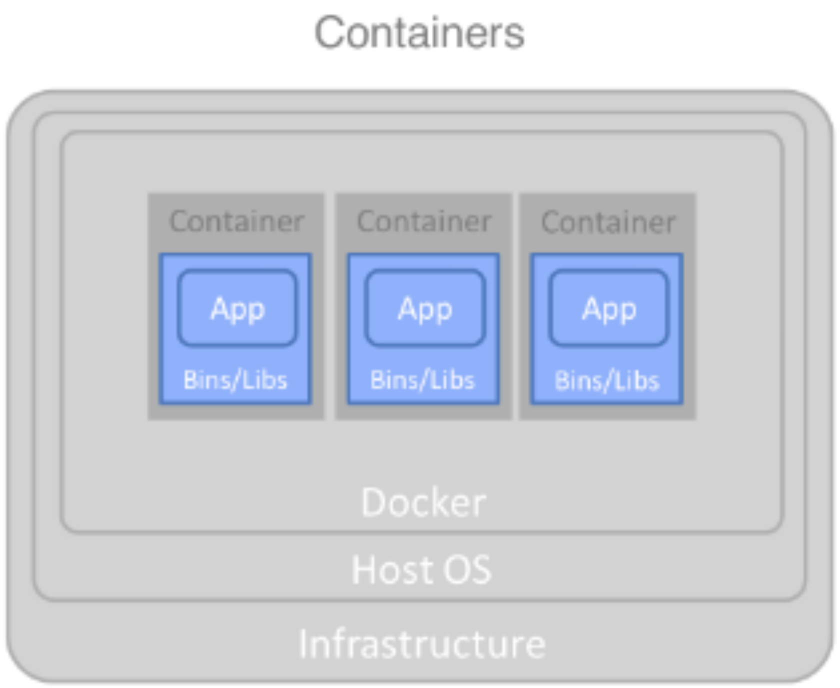
U.S. Patent No. 7,784,058 ("'058 Patent")

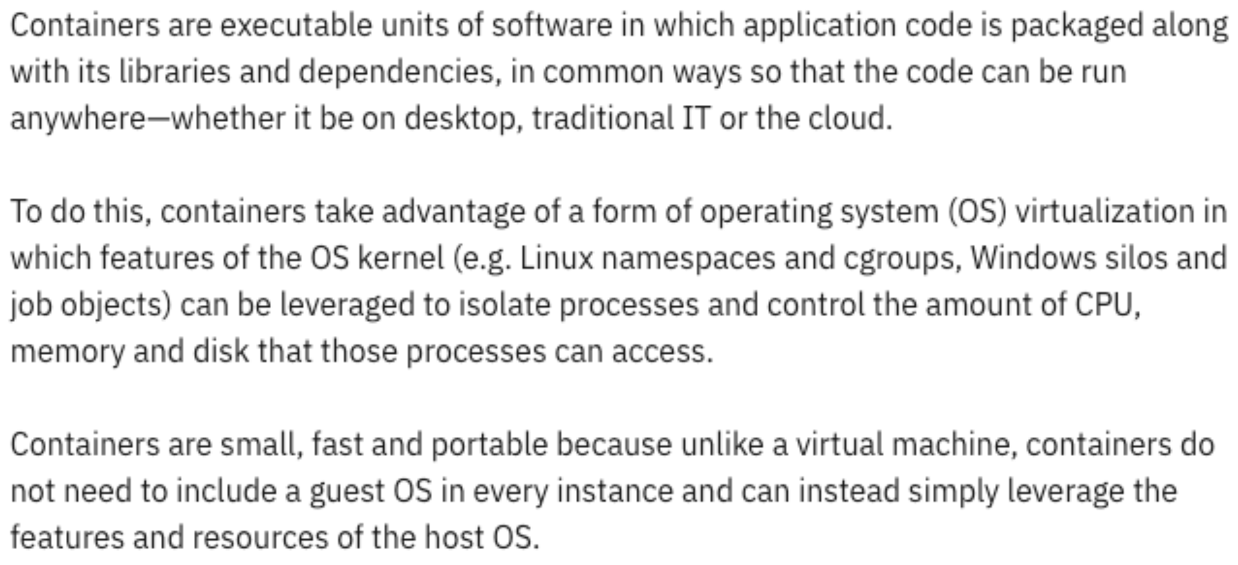
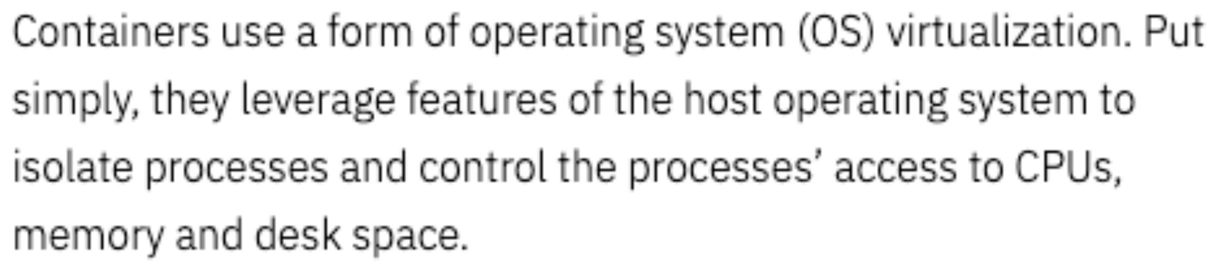
Accused Instrumentalities: IBM products and services using user mode critical system elements as shared libraries, including without limitation IBM Cloud Kubernetes Service (IKS), IBM Cloud Private (ICP), and IBM Hybrid Cloud mesh,, and all versions and variations thereof since the issuance of the asserted patent.

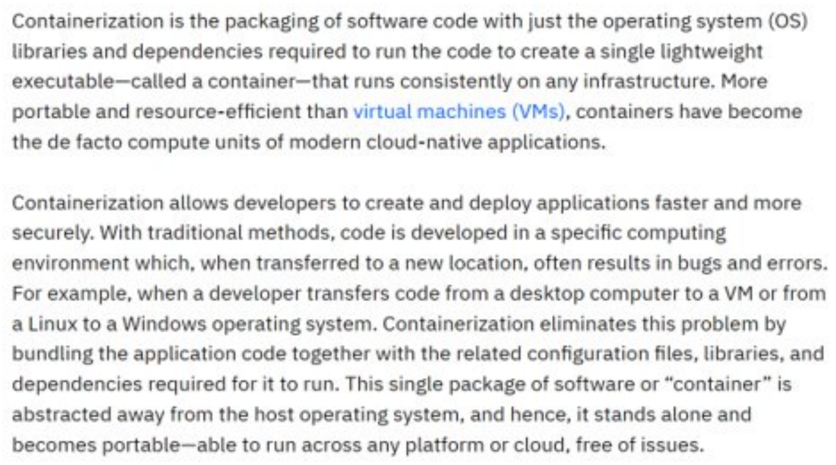
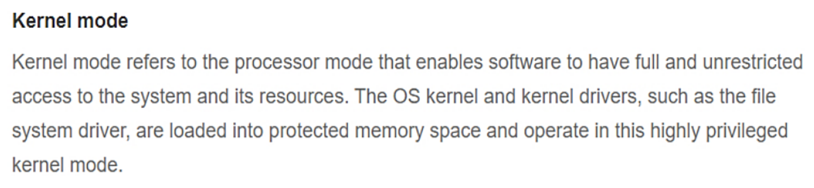
Each Accused Instrumentality infringes the claims in substantially the same way, and the evidence shown in this chart is similarly applicable to each Accused Instrumentality. Each claim limitation is literally infringed by each Accused Instrumentality. However, to the extent any claim limitation is not met literally, it is nonetheless met under the doctrine of equivalents because the differences between the claim limitation and each Accused Instrumentality would be insubstantial, and each Accused Instrumentality performs substantially the same function, in substantially the same way, to achieve the same result as the claimed invention. Notably, Defendant has not yet articulated which, if any, particular claim limitations it believes are not met by the Accused Instrumentalities.

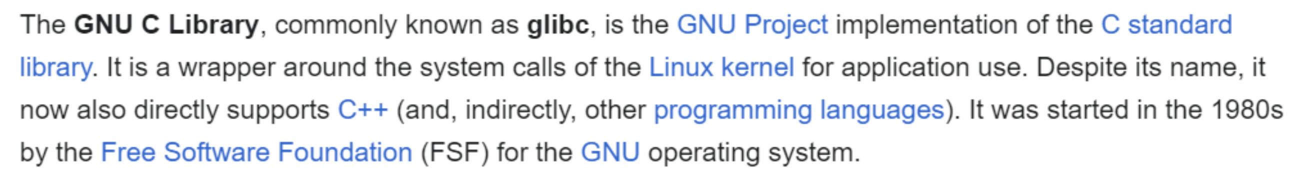
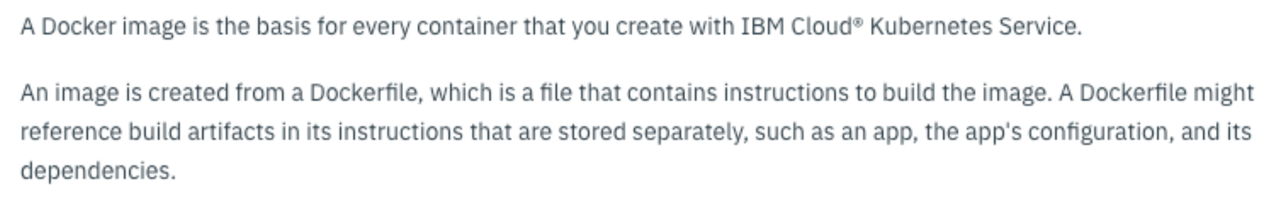
Claim 1

Claim 1	Accused Instrumentalities
[1pre] 1. A computing system for executing a plurality of software applications comprising:	To the extent the preamble is limiting, each Accused Instrumentality comprises or constitutes a computing system for executing a plurality of software applications as claimed. *See* claim limitations below. *See also, e.g.*: IBM Cloud® Kubernetes Service provides a fully managed container service for Docker (OCI) containers, so clients can deploy containerized apps onto a pool of compute hosts and subsequently manage those containers. Containers are automatically scheduled and placed onto available compute hosts based on your requirements and availability in the cluster. https://www.ibm.com/products/kubernetes-service With IBM Cloud Kubernetes Service, you can deploy Docker containers into pods that run on your worker nodes. The worker nodes come with a set of add-on pods to help you manage your containers. Install more add-ons through Helm, a Kubernetes package manager. These add-ons can extend your apps with dashboards, logging, IBM Cloud and IBM Watson® services and more. https://www.ibm.com/products/kubernetes-service

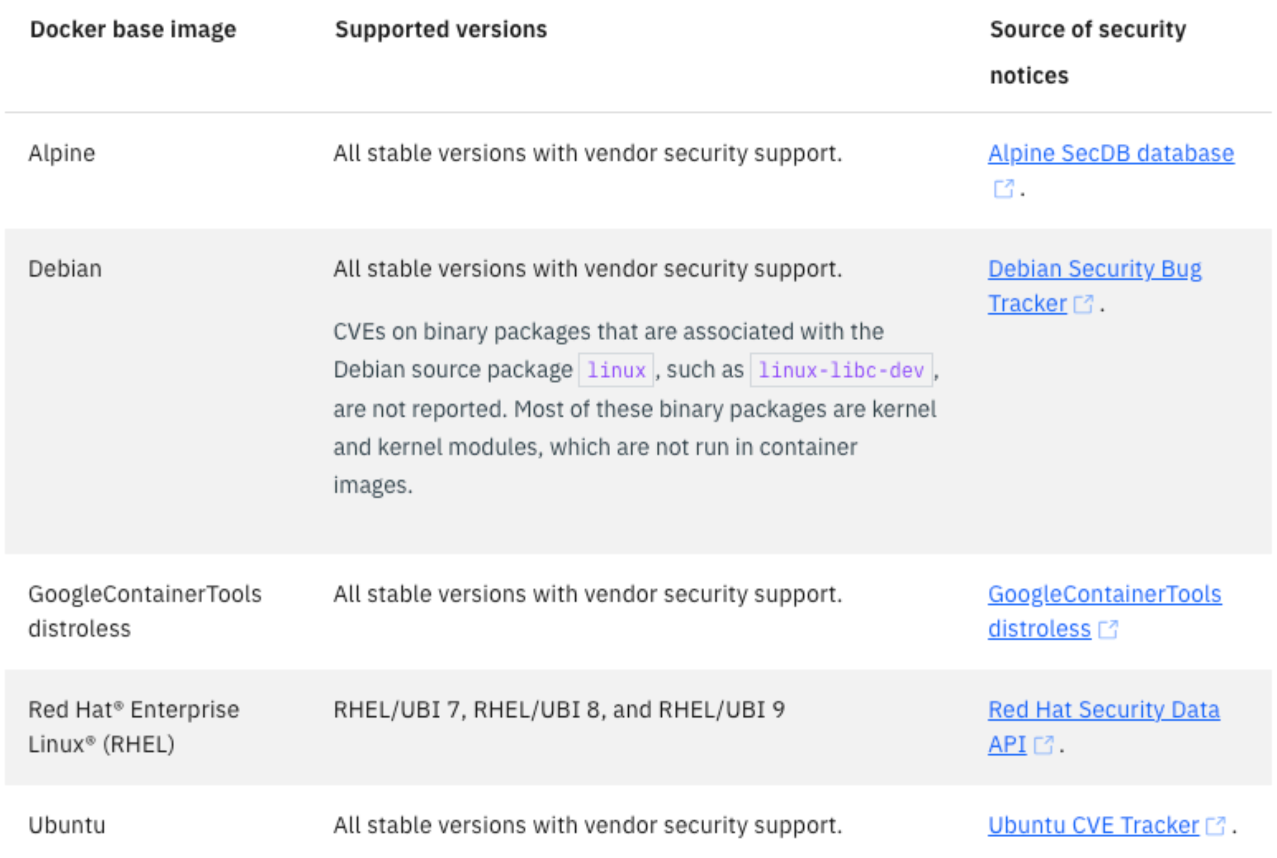
Claim 1	Accused Instrumentalities
	 Containers Container App Bins/Libs Container App Bins/Libs Container App Bins/Libs Docker Host OS Infrastructure https://developer.ibm.com/articles/true-benefits-of-moving-to-containers-1/
[1a] a) a processor;	Each Accused Instrumentality comprises a processor. *See, e.g.*:

Claim 1	Accused Instrumentalities
	 Containers are executable units of software in which application code is packaged along with its libraries and dependencies, in common ways so that the code can be run anywhere—whether it be on desktop, traditional IT or the cloud. To do this, containers take advantage of a form of operating system (OS) virtualization in which features of the OS kernel (e.g. Linux namespaces and cgroups, Windows silos and job objects) can be leveraged to isolate processes and control the amount of CPU, memory and disk that those processes can access. Containers are small, fast and portable because unlike a virtual machine, containers do not need to include a guest OS in every instance and can instead simply leverage the features and resources of the host OS. https://www.ibm.com/topics/containers  Containers use a form of operating system (OS) virtualization. Put simply, they leverage features of the host operating system to isolate processes and control the processes' access to CPUs, memory and desk space. https://www.ibm.com/blog/containers-vs-vms/

Claim 1	Accused Instrumentalities
[1b] b) an operating system having an operating system kernel having OS critical system elements (OSCSEs) for running in kernel mode using said processor; and,	Each Accused Instrumentality comprises an operating system having an operating system kernel having OS critical system elements (OSCSEs) for running in kernel mode using said processor. *See, e.g.*:  Containerization is the packaging of software code with just the operating system (OS) libraries and dependencies required to run the code to create a single lightweight executable—called a container—that runs consistently on any infrastructure. More portable and resource-efficient than virtual machines (VMs), containers have become the de facto compute units of modern cloud-native applications. Containerization allows developers to create and deploy applications faster and more securely. With traditional methods, code is developed in a specific computing environment which, when transferred to a new location, often results in bugs and errors. For example, when a developer transfers code from a desktop computer to a VM or from a Linux to a Windows operating system. Containerization eliminates this problem by bundling the application code together with the related configuration files, libraries, and dependencies required for it to run. This single package of software or "container" is abstracted away from the host operating system, and hence, it stands alone and becomes portable—able to run across any platform or cloud, free of issues. https://www.ibm.com/topics/containerization  **Kernel mode** Kernel mode refers to the processor mode that enables software to have full and unrestricted access to the system and its resources. The OS kernel and kernel drivers, such as the file system driver, are loaded into protected memory space and operate in this highly privileged kernel mode. https://www.techtarget.com/searchdatacenter/definition/kernel

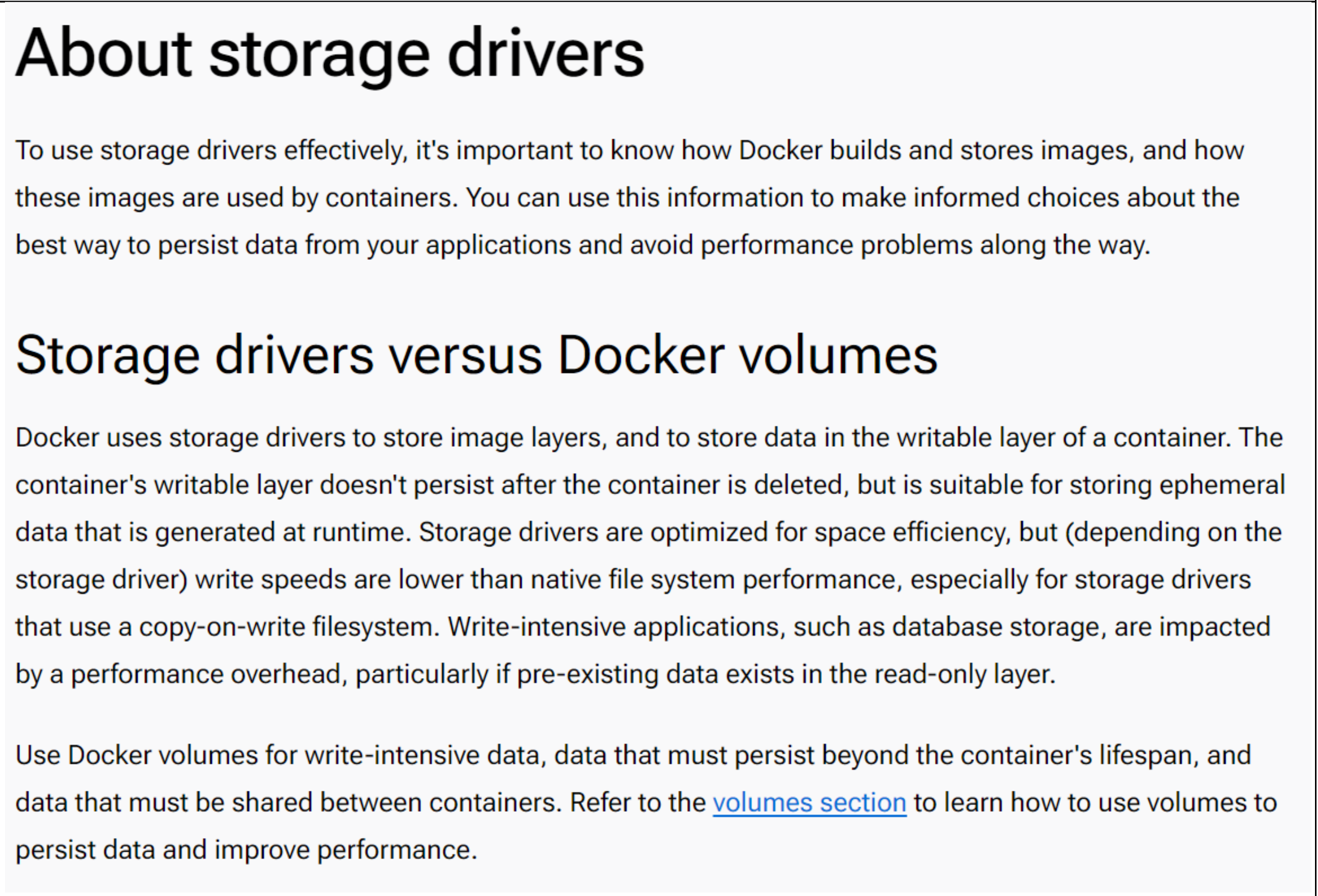
Claim 1	Accused Instrumentalities
	 The **GNU C Library**, commonly known as **glibc**, is the GNU Project implementation of the C standard library. It is a wrapper around the system calls of the Linux kernel for application use. Despite its name, it now also directly supports C++ (and, indirectly, other programming languages). It was started in the 1980s by the Free Software Foundation (FSF) for the GNU operating system. https://en.wikipedia.org/wiki/Glibc
[1c] c) a shared library having shared library critical system elements (SLCSEs) stored therein for use by the plurality of software applications in user mode and	Each Accused Instrumentality comprises a shared library having shared library critical system elements (SLCSEs) stored therein for use by the plurality of software applications in user mode. *See, e.g.*:  A Docker image is the basis for every container that you create with IBM Cloud® Kubernetes Service. An image is created from a Dockerfile, which is a file that contains instructions to build the image. A Dockerfile might reference build artifacts in its instructions that are stored separately, such as an app, the app's configuration, and its dependencies. https://cloud.ibm.com/docs/containers?topic=containers-images

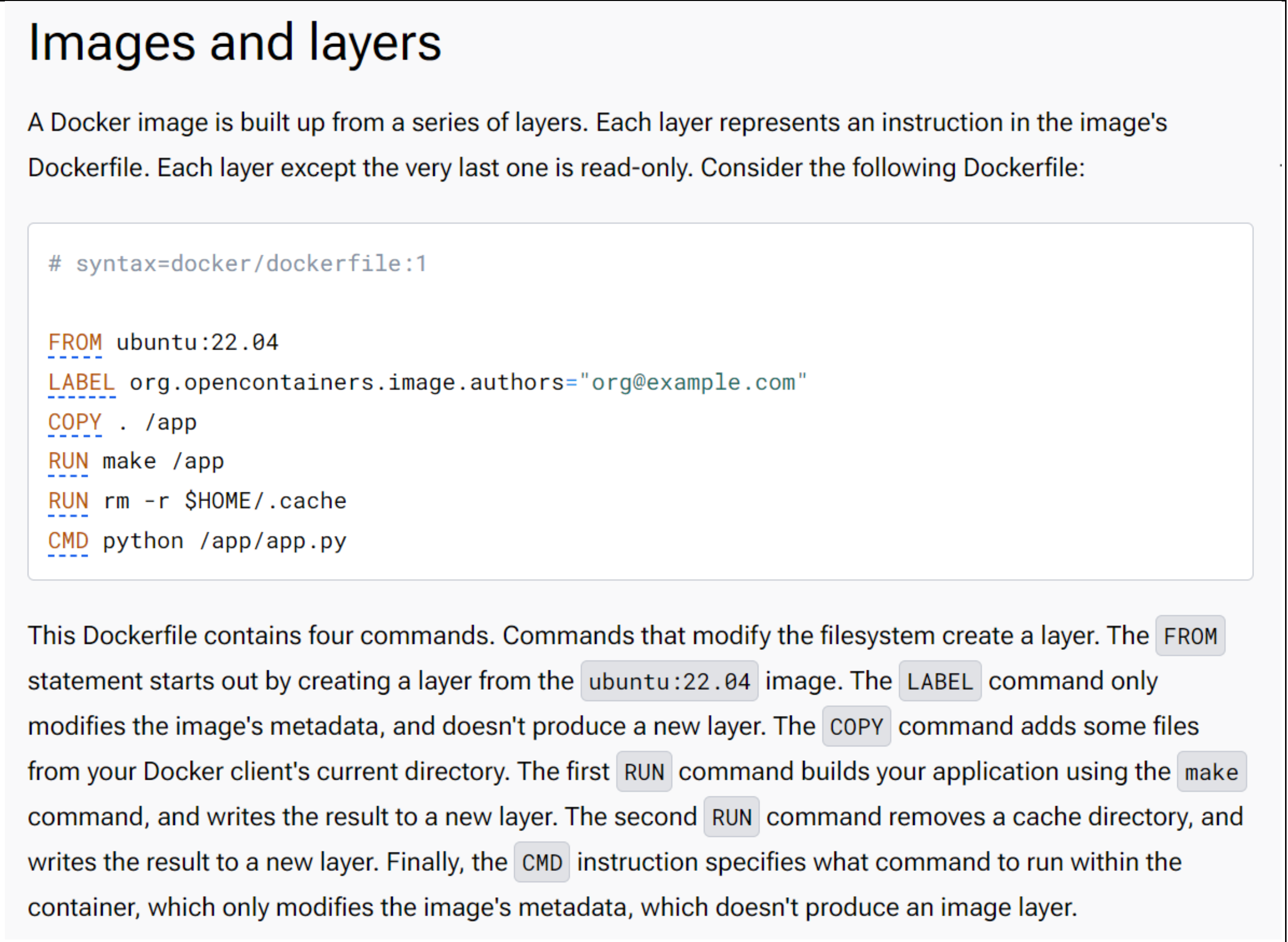
Claim 1	Accused Instrumentalities



Docker base image	Supported versions	Source of security notices
Alpine	All stable versions with vendor security support.	Alpine SecDB database.
Debian	All stable versions with vendor security support. CVEs on binary packages that are associated with the Debian source package `linux`, such as `linux-libc-dev`, are not reported. Most of these binary packages are kernel and kernel modules, which are not run in container images.	Debian Security Bug Tracker.
GoogleContainerTools distroless	All stable versions with vendor security support.	GoogleContainerTools distroless
Red Hat® Enterprise Linux® (RHEL)	RHEL/UBI 7, RHEL/UBI 8, and RHEL/UBI 9	Red Hat Security Data API.
Ubuntu	All stable versions with vendor security support.	Ubuntu CVE Tracker.

https://cloud.ibm.com/docs/Registry?topic=Registry-va_index&interface=ui

Claim 1	Accused Instrumentalities
	 **About storage drivers** To use storage drivers effectively, it's important to know how Docker builds and stores images, and how these images are used by containers. You can use this information to make informed choices about the best way to persist data from your applications and avoid performance problems along the way. **Storage drivers versus Docker volumes** Docker uses storage drivers to store image layers, and to store data in the writable layer of a container. The container's writable layer doesn't persist after the container is deleted, but is suitable for storing ephemeral data that is generated at runtime. Storage drivers are optimized for space efficiency, but (depending on the storage driver) write speeds are lower than native file system performance, especially for storage drivers that use a copy-on-write filesystem. Write-intensive applications, such as database storage, are impacted by a performance overhead, particularly if pre-existing data exists in the read-only layer. Use Docker volumes for write-intensive data, data that must persist beyond the container's lifespan, and data that must be shared between containers. Refer to the volumes section to learn how to use volumes to persist data and improve performance. https://docs.docker.com/storage/storagedriver/

Claim 1	Accused Instrumentalities
	 https://docs.docker.com/storage/storagedriver/

Images and layers

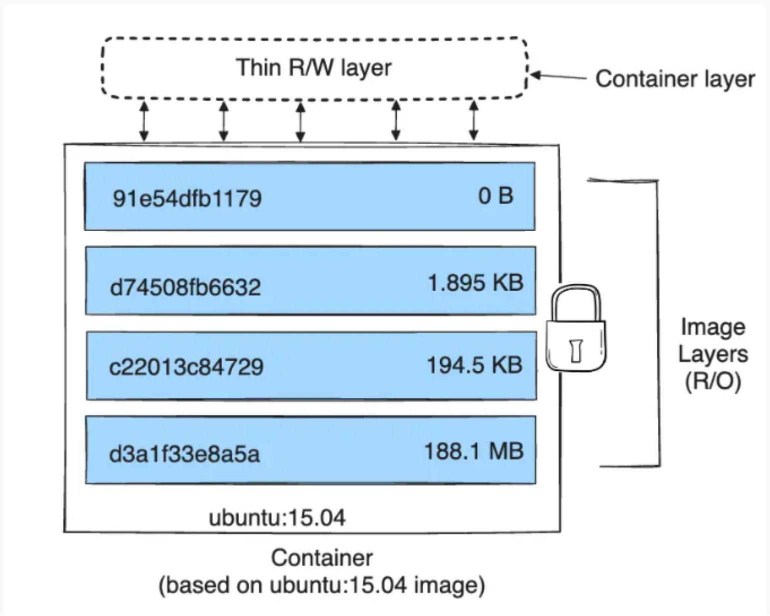
A Docker image is built up from a series of layers. Each layer represents an instruction in the image's Dockerfile. Each layer except the very last one is read-only. Consider the following Dockerfile:

```
# syntax=docker/dockerfile:1

FROM ubuntu:22.04
LABEL org.opencontainers.image.authors="org@example.com"
COPY . /app
RUN make /app
RUN rm -r $HOME/.cache
CMD python /app/app.py
```

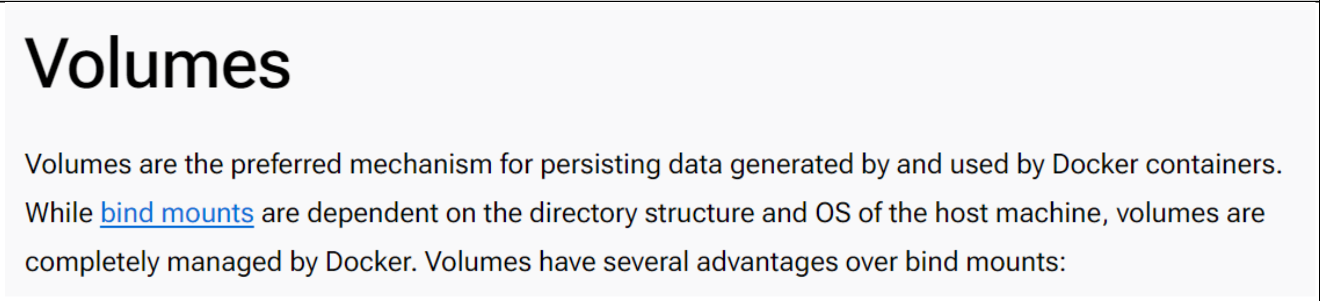
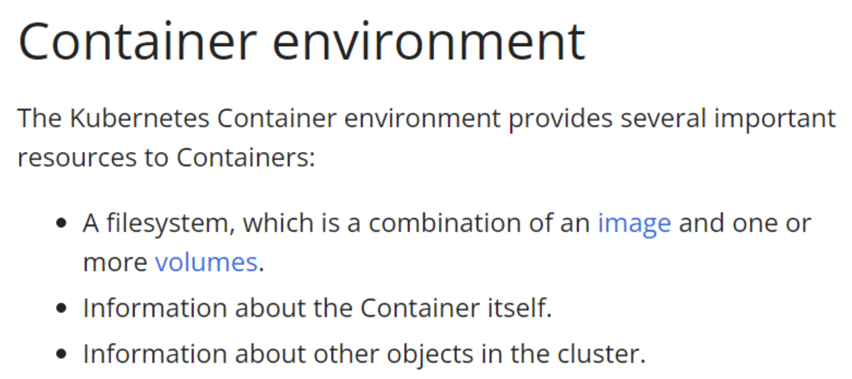
This Dockerfile contains four commands. Commands that modify the filesystem create a layer. The `FROM` statement starts out by creating a layer from the `ubuntu:22.04` image. The `LABEL` command only modifies the image's metadata, and doesn't produce a new layer. The `COPY` command adds some files from your Docker client's current directory. The first `RUN` command builds your application using the `make` command, and writes the result to a new layer. The second `RUN` command removes a cache directory, and writes the result to a new layer. Finally, the `CMD` instruction specifies what command to run within the container, which only modifies the image's metadata, which doesn't produce an image layer.

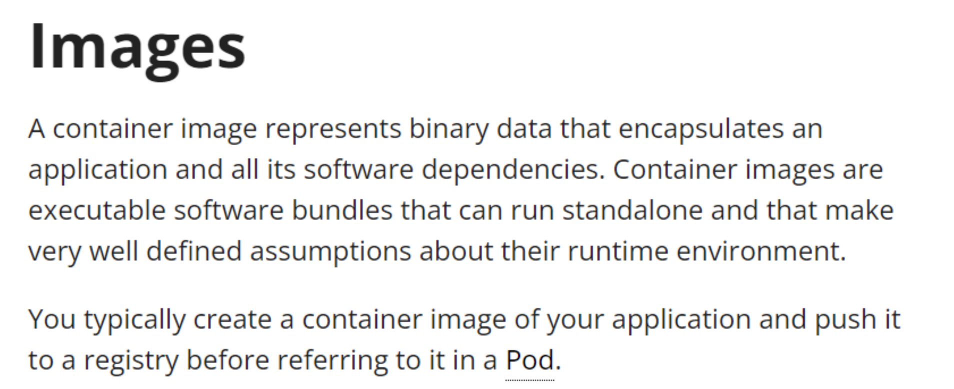
Each layer is only a set of differences from the layer before it. Note that both *adding*, and *removing* files will result in a new layer. In the example above, the `$HOME/.cache` directory is removed, but will still be available in the previous layer and add up to the image's total size. Refer to the Best practices for writing Dockerfiles and use multi-stage builds sections to learn how to optimize your Dockerfiles for efficient images.

The layers are stacked on top of each other. When you create a new container, you add a new writable layer on top of the underlying layers. This layer is often called the "container layer". All changes made to the running container, such as writing new files, modifying existing files, and deleting files, are written to this thin writable container layer. The diagram below shows a container based on an `ubuntu:15.04` image.



https://docs.docker.com/storage/storagedriver/

Claim 1	Accused Instrumentalities
	 Volumes Volumes are the preferred mechanism for persisting data generated by and used by Docker containers. While bind mounts are dependent on the directory structure and OS of the host machine, volumes are completely managed by Docker. Volumes have several advantages over bind mounts: https://kubernetes.io/docs/concepts/storage/volumes/  Container environment The Kubernetes Container environment provides several important resources to Containers: • A filesystem, which is a combination of an image and one or more volumes. • Information about the Container itself. • Information about other objects in the cluster. https://kubernetes.io/docs/concepts/containers/container-environment/

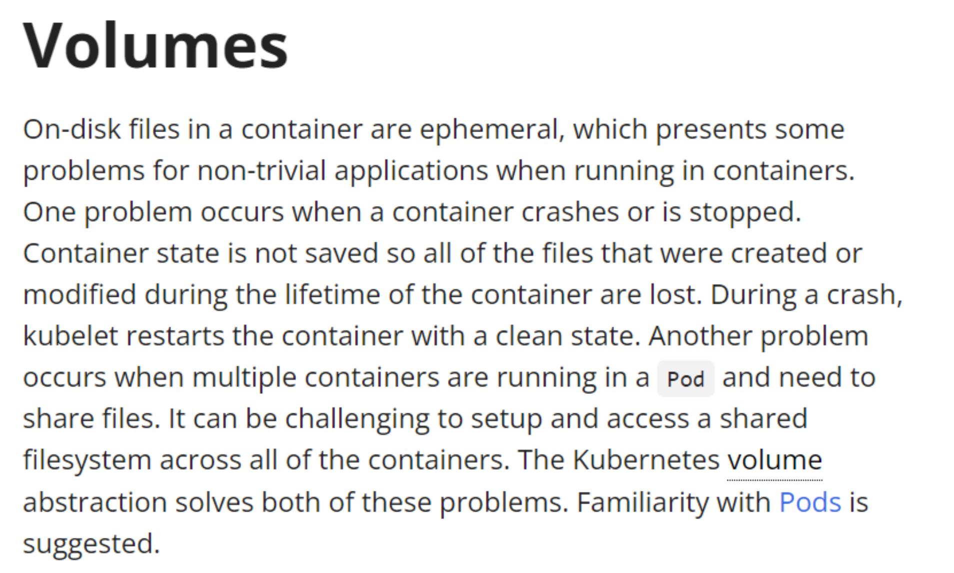
<table>
<tr><th>Claim 1</th><th>Accused Instrumentalities</th></tr>
<tr><td></td><td>

Images

A container image represents binary data that encapsulates an application and all its software dependencies. Container images are executable software bundles that can run standalone and that make very well defined assumptions about their runtime environment.

You typically create a container image of your application and push it to a registry before referring to it in a Pod.

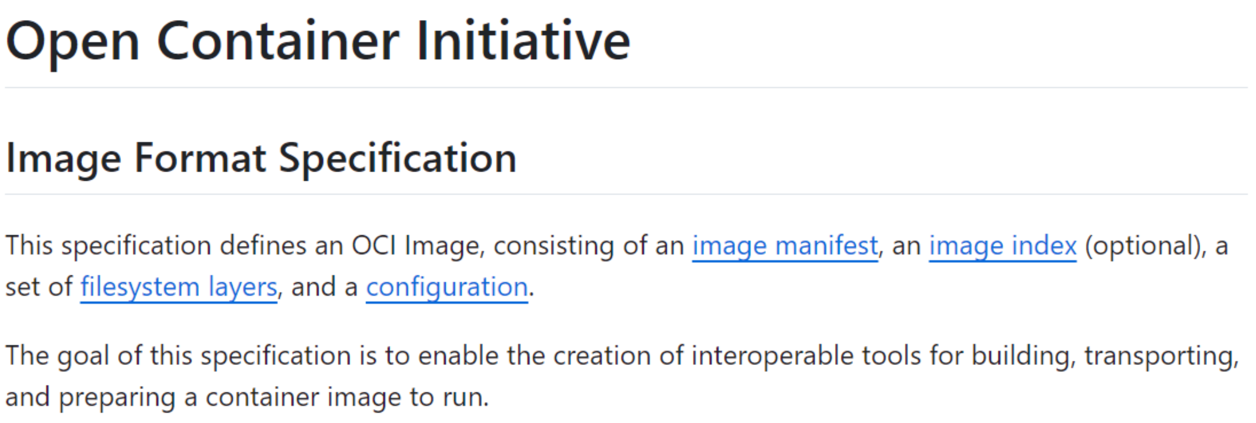
https://kubernetes.io/docs/concepts/containers/images/



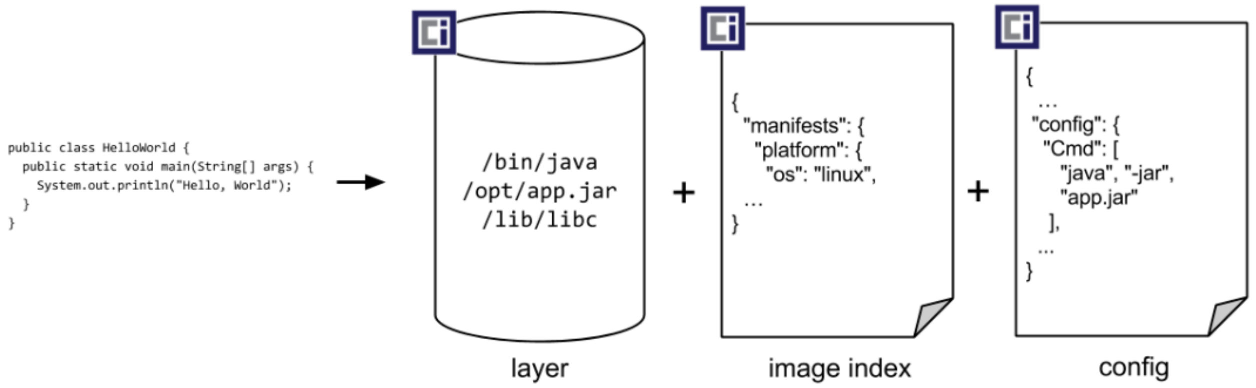
Volumes

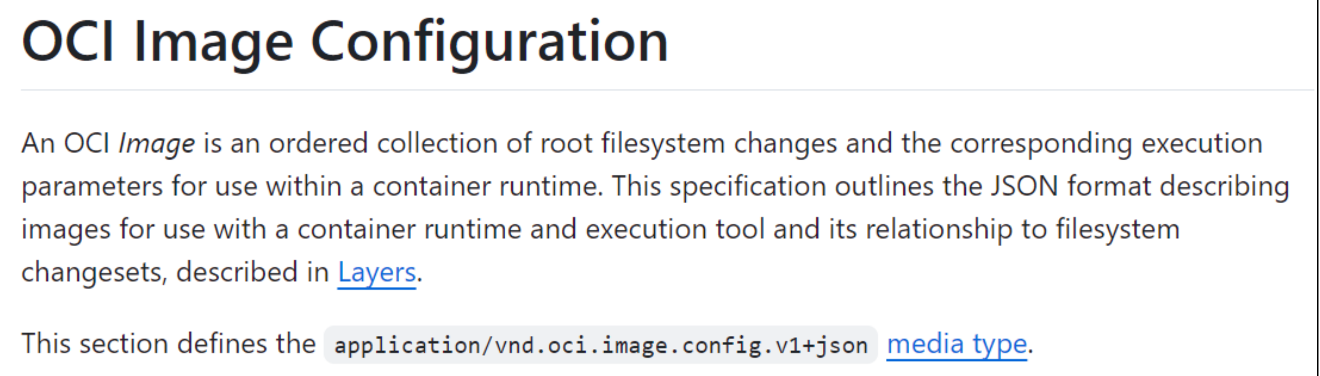
On-disk files in a container are ephemeral, which presents some problems for non-trivial applications when running in containers. One problem occurs when a container crashes or is stopped. Container state is not saved so all of the files that were created or modified during the lifetime of the container are lost. During a crash, kubelet restarts the container with a clean state. Another problem occurs when multiple containers are running in a `Pod` and need to share files. It can be challenging to setup and access a shared filesystem across all of the containers. The Kubernetes volume abstraction solves both of these problems. Familiarity with Pods is suggested.

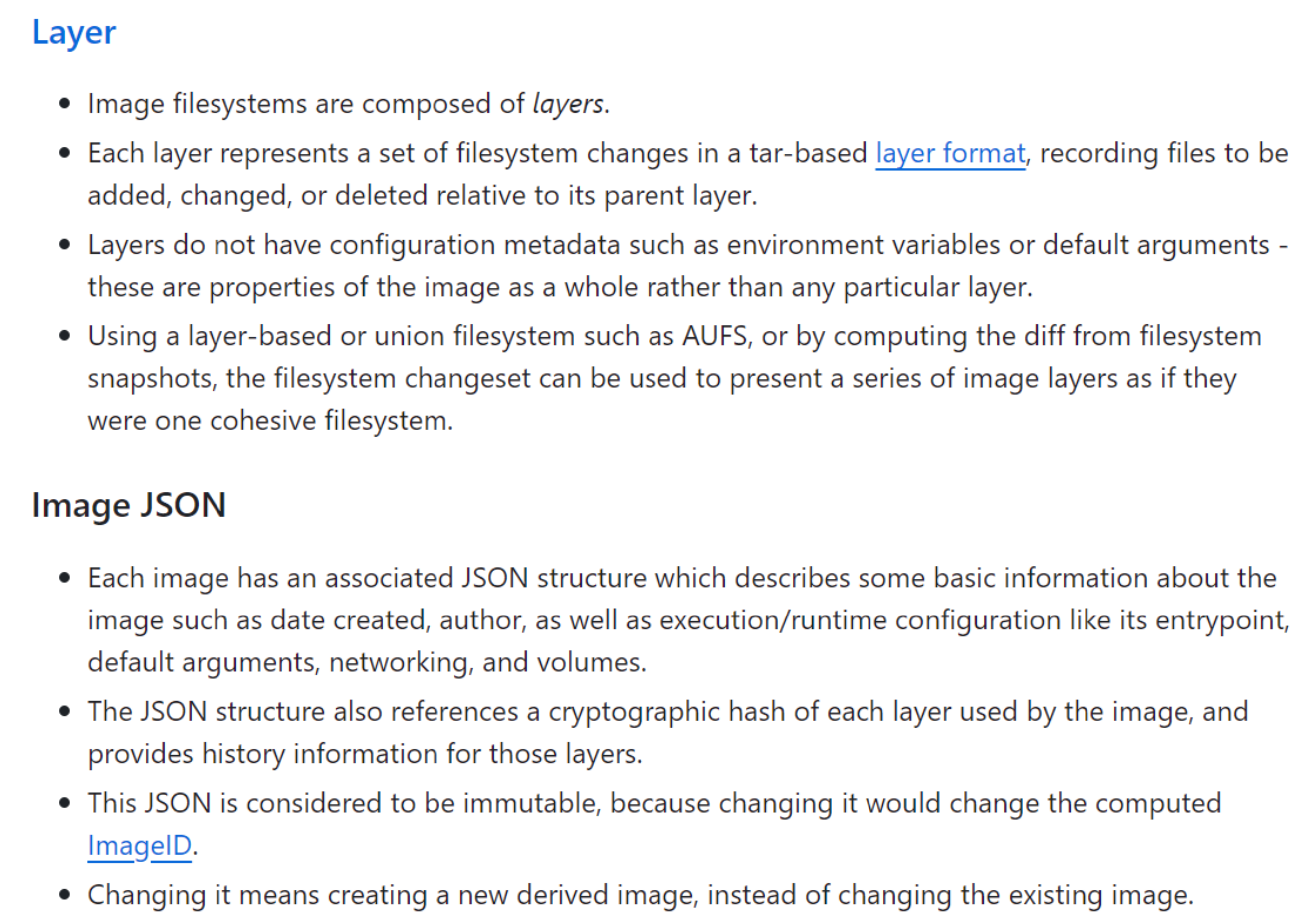
https://kubernetes.io/docs/concepts/storage/volumes/</td></tr>
</table>

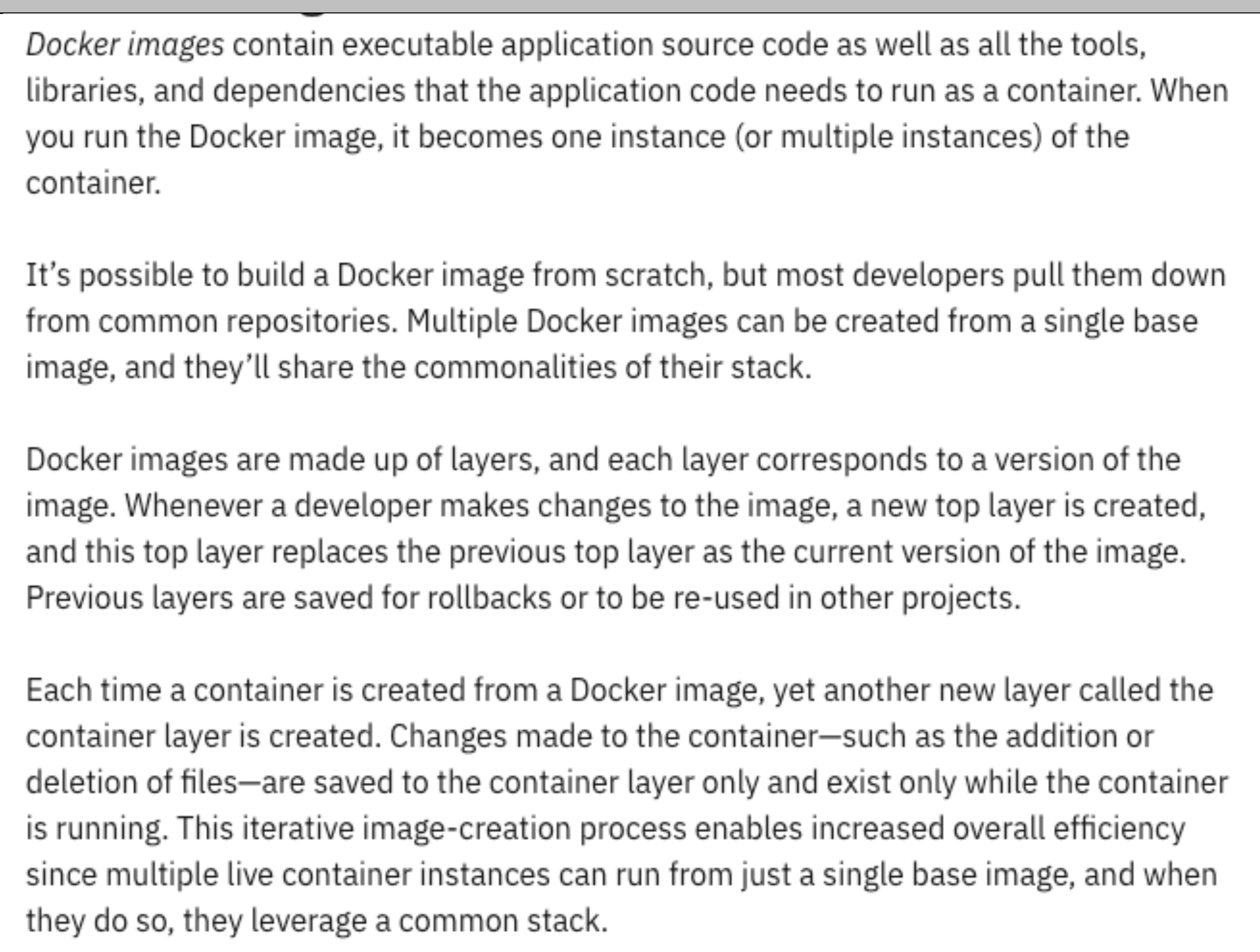
Claim 1	Accused Instrumentalities
	 **Open Container Initiative** **Image Format Specification** This specification defines an OCI Image, consisting of an image manifest, an image index (optional), a set of filesystem layers, and a configuration. The goal of this specification is to enable the creation of interoperable tools for building, transporting, and preparing a container image to run. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/spec.md

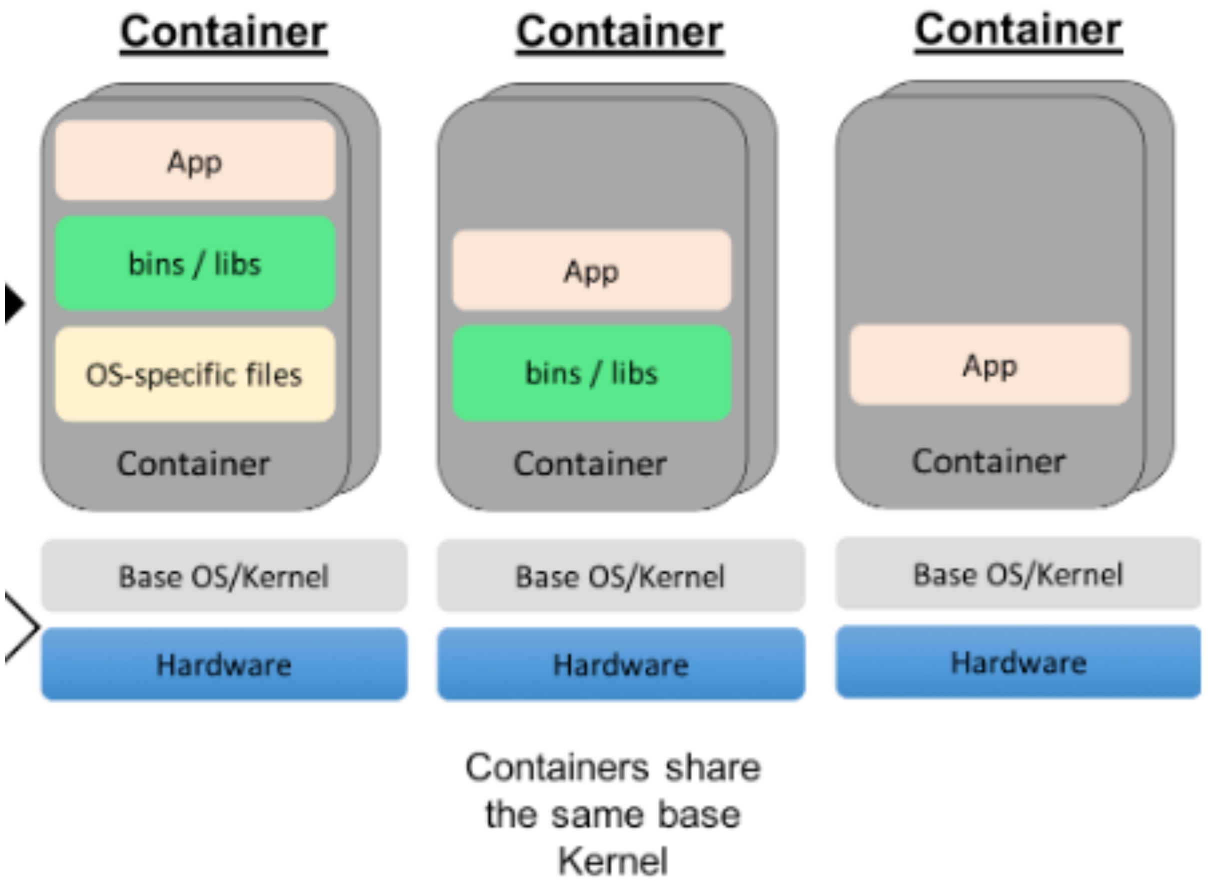
Claim 1	Accused Instrumentalities

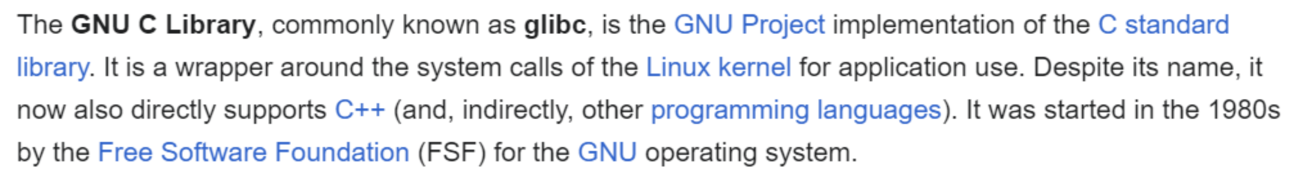
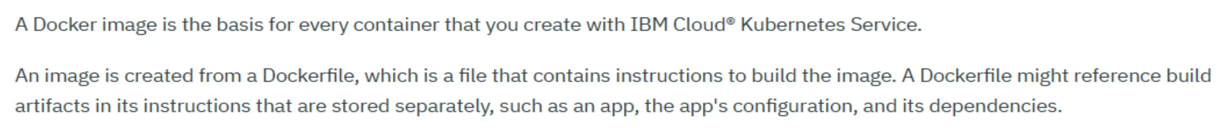
	Overview
	
	At a high level the image manifest contains metadata about the contents and dependencies of the image including the content-addressable identity of one or more filesystem layer changeset archives that will be unpacked to make up the final runnable filesystem. The image configuration includes information such as application arguments, environments, etc. The image index is a higher-level manifest which points to a list of manifests and descriptors. Typically, these manifests may provide different implementations of the image, possibly varying by platform or other attributes.
	
	 public class HelloWorld {
	public static void main(String[] args) {
	System.out.println("Hello, World");
	}
	}
	/bin/java
	/opt/app.jar
	/lib/libc
	+
	{
	"manifests": {
	"platform": {
	"os": "linux",
	...
	}
	+
	{
	...
	"config": {
	"Cmd": [
	"java", "-jar",
	"app.jar"
	],
	...
	}
	layer
	image index
	config
	
	https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/spec.md

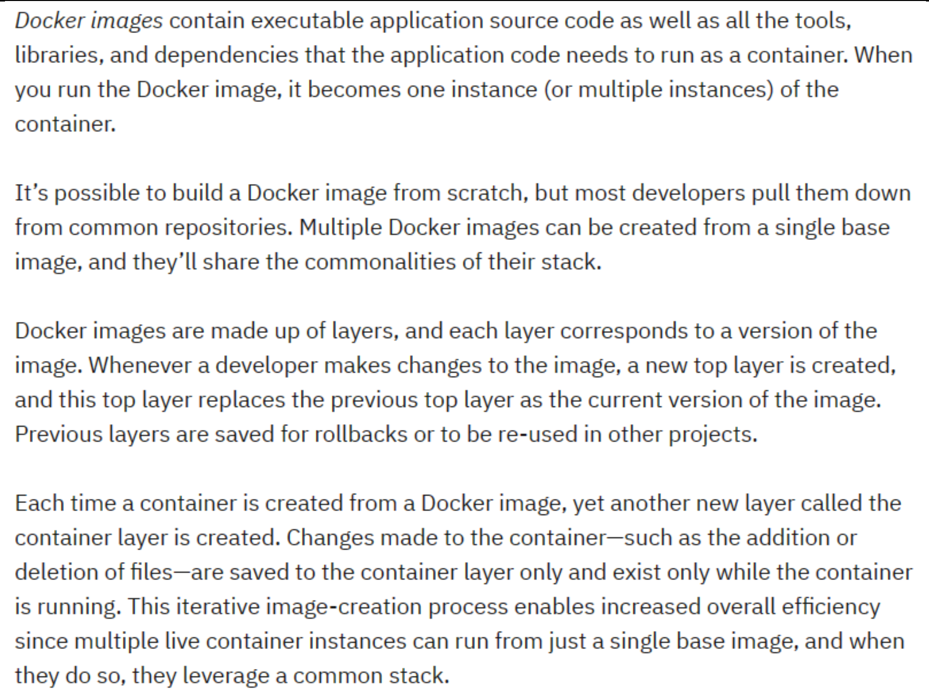
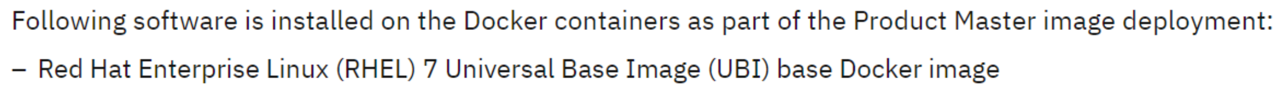
Claim 1	Accused Instrumentalities
	 OCI Image Configuration An OCI *Image* is an ordered collection of root filesystem changes and the corresponding execution parameters for use within a container runtime. This specification outlines the JSON format describing images for use with a container runtime and execution tool and its relationship to filesystem changesets, described in Layers. This section defines the `application/vnd.oci.image.config.v1+json` media type. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/config.md

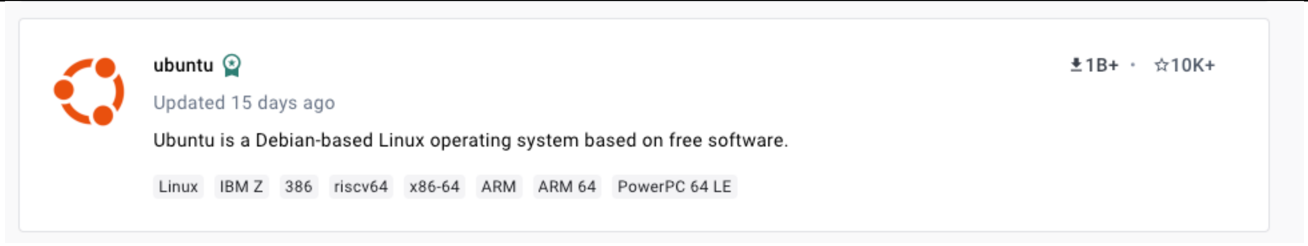
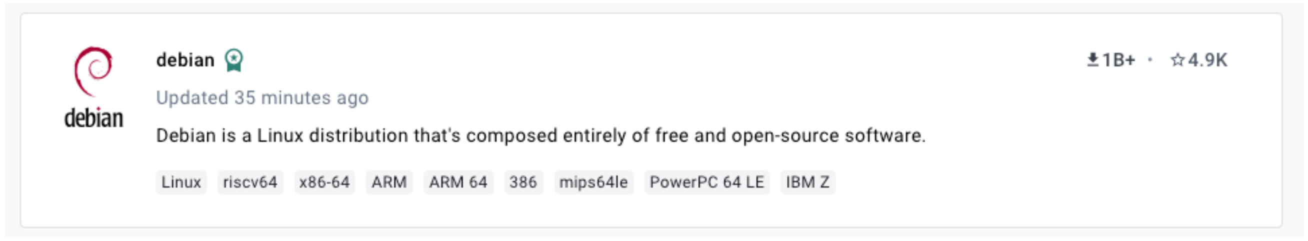
Claim 1	Accused Instrumentalities
	 **Layer** • Image filesystems are composed of *layers*. • Each layer represents a set of filesystem changes in a tar-based layer format, recording files to be added, changed, or deleted relative to its parent layer. • Layers do not have configuration metadata such as environment variables or default arguments - these are properties of the image as a whole rather than any particular layer. • Using a layer-based or union filesystem such as AUFS, or by computing the diff from filesystem snapshots, the filesystem changeset can be used to present a series of image layers as if they were one cohesive filesystem. **Image JSON** • Each image has an associated JSON structure which describes some basic information about the image such as date created, author, as well as execution/runtime configuration like its entrypoint, default arguments, networking, and volumes. • The JSON structure also references a cryptographic hash of each layer used by the image, and provides history information for those layers. • This JSON is considered to be immutable, because changing it would change the computed ImageID. • Changing it means creating a new derived image, instead of changing the existing image. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/config.md

Claim 1	Accused Instrumentalities
	 *Docker images* contain executable application source code as well as all the tools, libraries, and dependencies that the application code needs to run as a container. When you run the Docker image, it becomes one instance (or multiple instances) of the container. It's possible to build a Docker image from scratch, but most developers pull them down from common repositories. Multiple Docker images can be created from a single base image, and they'll share the commonalities of their stack. Docker images are made up of layers, and each layer corresponds to a version of the image. Whenever a developer makes changes to the image, a new top layer is created, and this top layer replaces the previous top layer as the current version of the image. Previous layers are saved for rollbacks or to be re-used in other projects. Each time a container is created from a Docker image, yet another new layer called the container layer is created. Changes made to the container—such as the addition or deletion of files—are saved to the container layer only and exist only while the container is running. This iterative image-creation process enables increased overall efficiency since multiple live container instances can run from just a single base image, and when they do so, they leverage a common stack. https://www.ibm.com/topics/docker

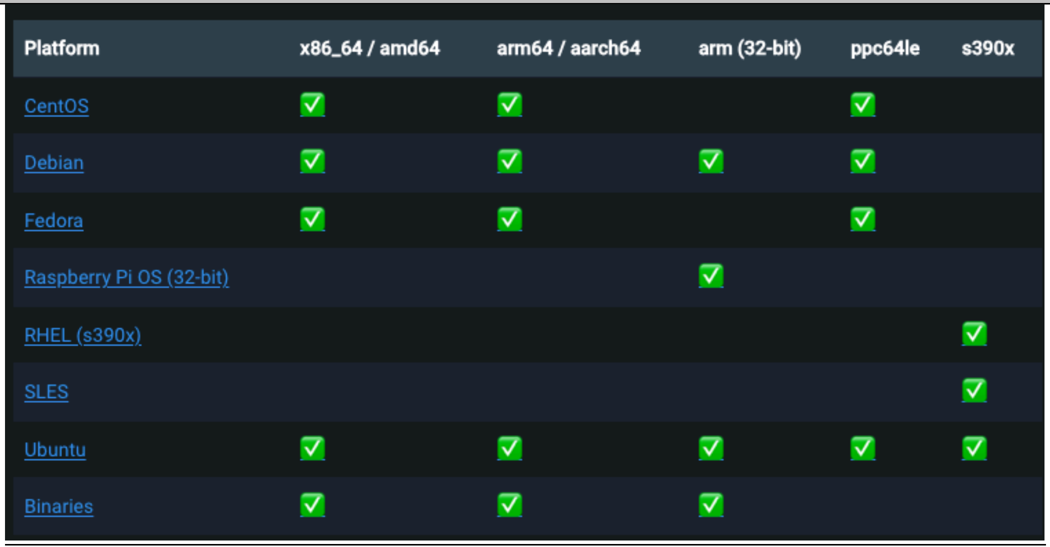
Claim 1	Accused Instrumentalities
	 Container Container Container App bins / libs OS-specific files Container App bins / libs Container App Container Base OS/Kernel Base OS/Kernel Base OS/Kernel Hardware Hardware Hardware Containers share the same base Kernel https://ibm.github.io/kube101/

Claim 1	Accused Instrumentalities
	 The **GNU C Library**, commonly known as **glibc**, is the GNU Project implementation of the C standard library. It is a wrapper around the system calls of the Linux kernel for application use. Despite its name, it now also directly supports C++ (and, indirectly, other programming languages). It was started in the 1980s by the Free Software Foundation (FSF) for the GNU operating system. https://en.wikipedia.org/wiki/Glibc
[1d] i) wherein some of the SLCSEs stored in the shared library are functional replicas of OSCSEs and are accessible to some of the plurality of software applications and when one of the SLCSEs is accessed by one or more of the plurality of software applications it forms a part of the one or more of the plurality of software applications,	In each Accused Instrumentality, some of the SLCSEs stored in the shared library are functional replicas of OSCSEs and are accessible to some of the plurality of software applications and when one of the SLCSEs is accessed by one or more of the plurality of software applications it forms a part of the one or more of the plurality of software applications. For example, a base image serves as a self-contained unit that encompasses all the necessary components for an application to run, including the application code, runtime environment, system tools, and dependencies (i.e., SLCSEs). The images are based on existing Linux distributions, such as Debian and Ubuntu, including essential system elements (i.e., functional replicas of OSCSEs). Each container image is based on a specific base image, which contains the application code, and dependencies, including system libraries or shared library critical system elements (SLCSEs). When the container runs the image, it creates a runtime instance of that container image. *See, e.g.*:  A Docker image is the basis for every container that you create with IBM Cloud® Kubernetes Service. An image is created from a Dockerfile, which is a file that contains instructions to build the image. A Dockerfile might reference build artifacts in its instructions that are stored separately, such as an app, the app's configuration, and its dependencies. https://cloud.ibm.com/docs/containers?topic=containers-images

Claim 1	Accused Instrumentalities
	 *Docker images* contain executable application source code as well as all the tools, libraries, and dependencies that the application code needs to run as a container. When you run the Docker image, it becomes one instance (or multiple instances) of the container. It's possible to build a Docker image from scratch, but most developers pull them down from common repositories. Multiple Docker images can be created from a single base image, and they'll share the commonalities of their stack. Docker images are made up of layers, and each layer corresponds to a version of the image. Whenever a developer makes changes to the image, a new top layer is created, and this top layer replaces the previous top layer as the current version of the image. Previous layers are saved for rollbacks or to be re-used in other projects. Each time a container is created from a Docker image, yet another new layer called the container layer is created. Changes made to the container—such as the addition or deletion of files—are saved to the container layer only and exist only while the container is running. This iterative image-creation process enables increased overall efficiency since multiple live container instances can run from just a single base image, and when they do so, they leverage a common stack. https://www.ibm.com/topics/docker  Following software is installed on the Docker containers as part of the Product Master image deployment: – Red Hat Enterprise Linux (RHEL) 7 Universal Base Image (UBI) base Docker image https://www.ibm.com/docs/en/product-master/12.0.0?topic=deployment-installing-product-by-using-docker-images

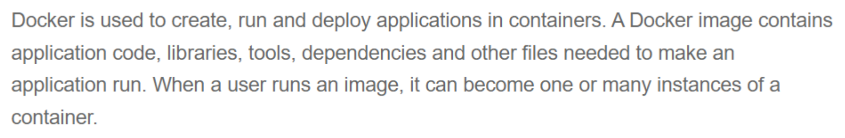
Claim 1	Accused Instrumentalities
	 ubuntu Updated 15 days ago ⭳1B+ · ☆10K+ Ubuntu is a Debian-based Linux operating system based on free software. Linux IBM Z 386 riscv64 x86-64 ARM ARM 64 PowerPC 64 LE  debian Updated 35 minutes ago ⭳1B+ · ☆4.9K Debian is a Linux distribution that's composed entirely of free and open-source software. Linux riscv64 x86-64 ARM ARM 64 386 mips64le PowerPC 64 LE IBM Z https://hub.docker.com/search?image_filter=official&type=image&q=

Claim 1	Accused Instrumentalities



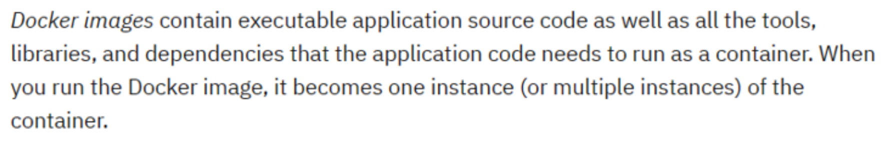
Platform	x86_64 / amd64	arm64 / aarch64	arm (32-bit)	ppc64le	s390x
CentOS	✓	✓		✓	
Debian	✓	✓	✓	✓	
Fedora	✓	✓		✓	
Raspberry Pi OS (32-bit)			✓		
RHEL (s390x)					✓
SLES					✓
Ubuntu	✓	✓	✓	✓	✓
Binaries	✓	✓	✓		

https://docs.docker.com/engine/install/



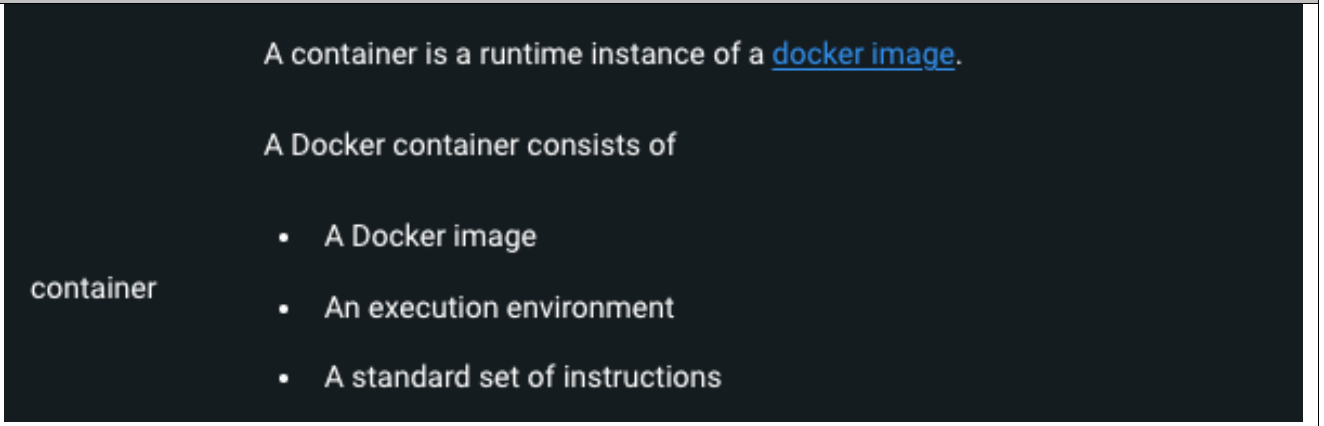
Docker is used to create, run and deploy applications in containers. A Docker image contains application code, libraries, tools, dependencies and other files needed to make an application run. When a user runs an image, it can become one or many instances of a container.

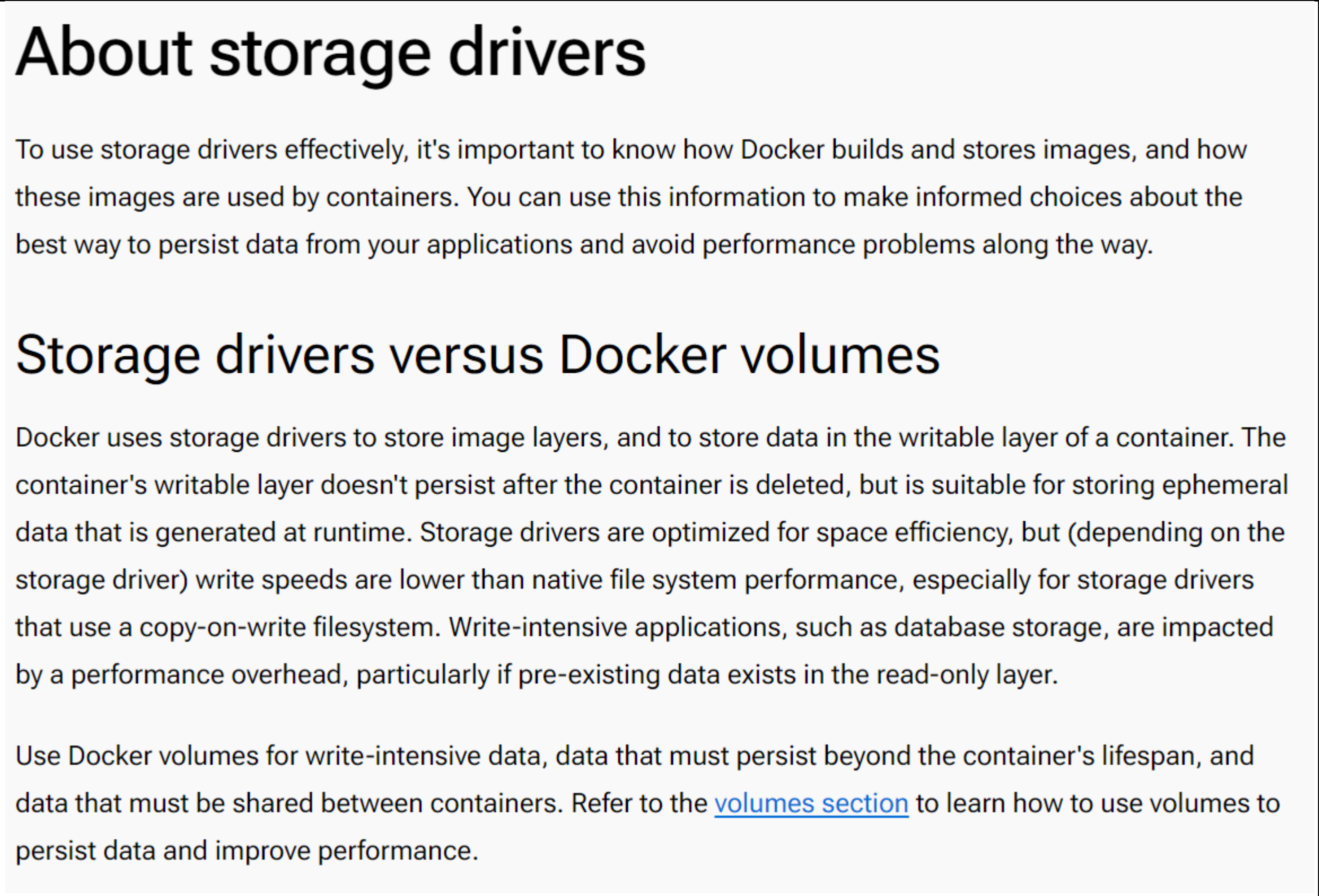
https://www.techtarget.com/searchitoperations/definition/Docker-image



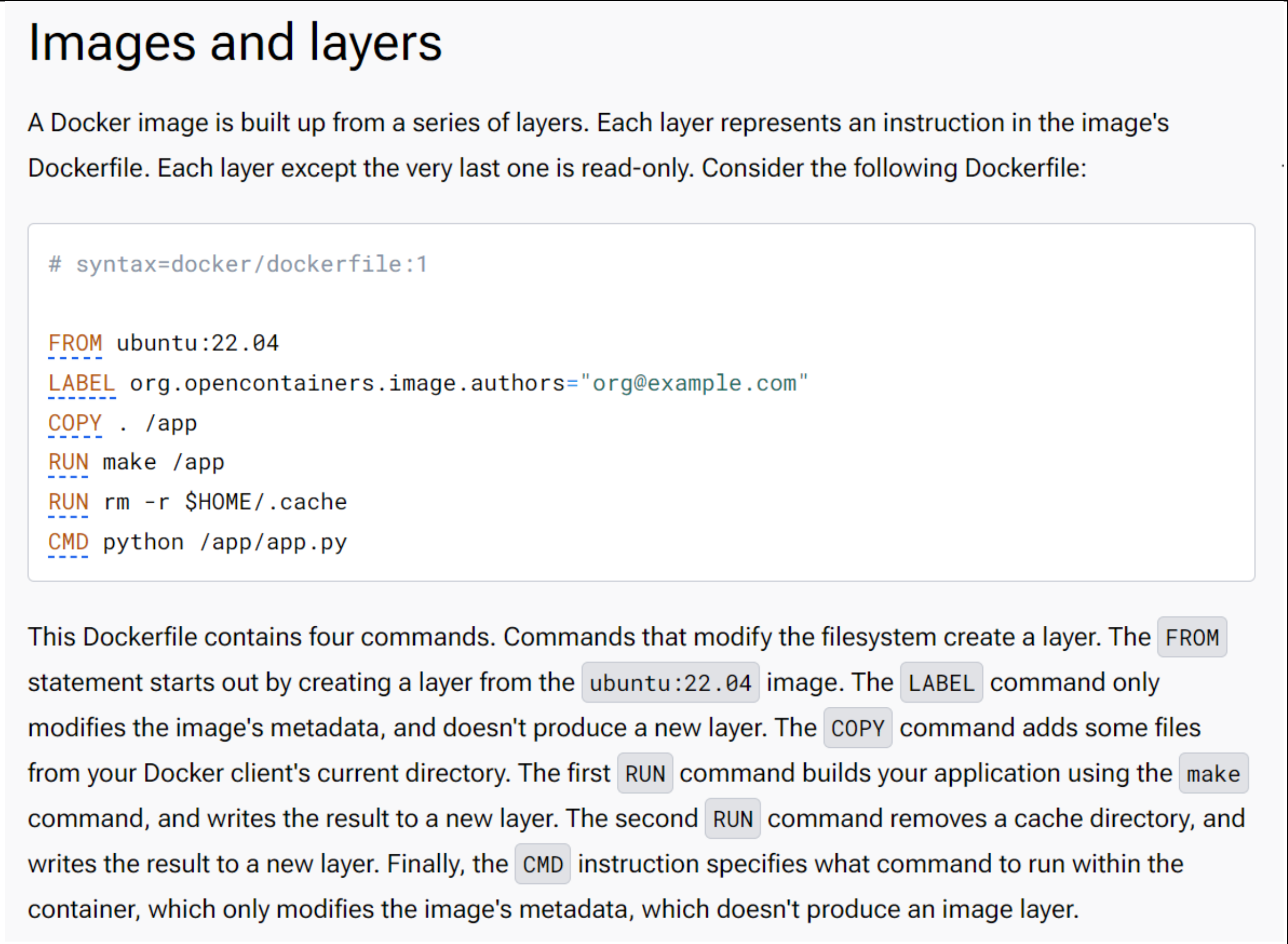
Docker images contain executable application source code as well as all the tools, libraries, and dependencies that the application code needs to run as a container. When you run the Docker image, it becomes one instance (or multiple instances) of the container.

https://www.ibm.com/topics/docker

Claim 1	Accused Instrumentalities
	 container A container is a runtime instance of a docker image. A Docker container consists of: • A Docker image • An execution environment • A standard set of instructions https://docs.docker.com/glossary/#image

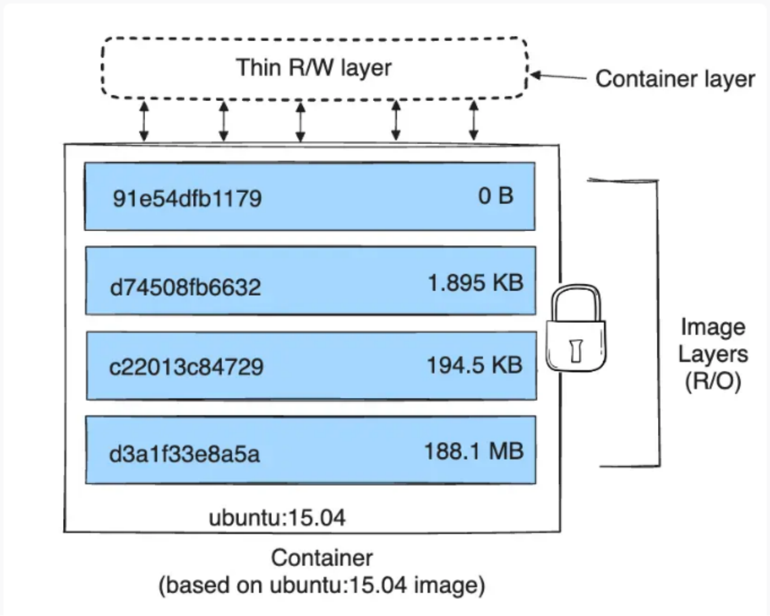
Claim 1	Accused Instrumentalities
	 ## About storage drivers To use storage drivers effectively, it's important to know how Docker builds and stores images, and how these images are used by containers. You can use this information to make informed choices about the best way to persist data from your applications and avoid performance problems along the way. ## Storage drivers versus Docker volumes Docker uses storage drivers to store image layers, and to store data in the writable layer of a container. The container's writable layer doesn't persist after the container is deleted, but is suitable for storing ephemeral data that is generated at runtime. Storage drivers are optimized for space efficiency, but (depending on the storage driver) write speeds are lower than native file system performance, especially for storage drivers that use a copy-on-write filesystem. Write-intensive applications, such as database storage, are impacted by a performance overhead, particularly if pre-existing data exists in the read-only layer. Use Docker volumes for write-intensive data, data that must persist beyond the container's lifespan, and data that must be shared between containers. Refer to the volumes section to learn how to use volumes to persist data and improve performance. https://docs.docker.com/storage/storagedriver/

Claim 1	Accused Instrumentalities

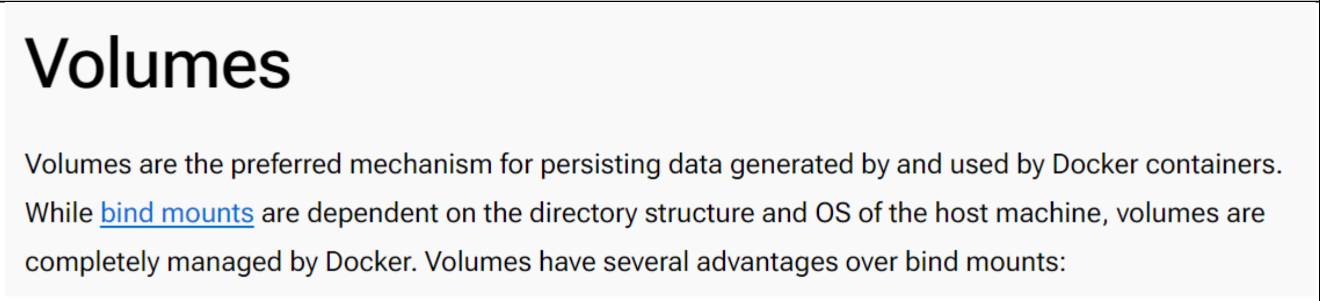
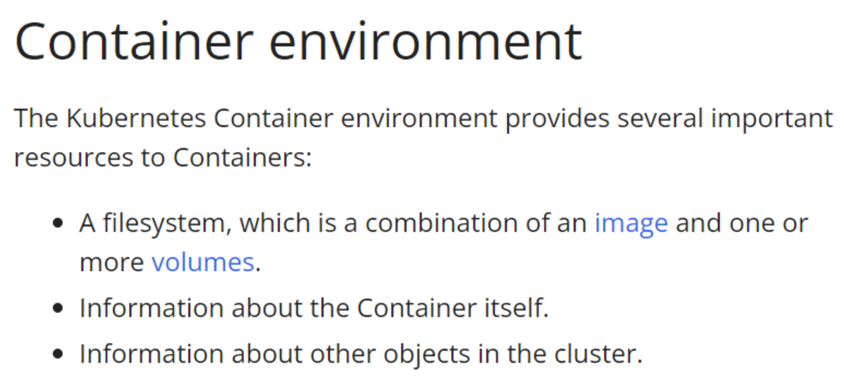
	
	Images and layers
	A Docker image is built up from a series of layers. Each layer represents an instruction in the image's Dockerfile. Each layer except the very last one is read-only. Consider the following Dockerfile:
	`# syntax=docker/dockerfile:1`
	
	`FROM ubuntu:22.04`
	`LABEL org.opencontainers.image.authors="org@example.com"`
	`COPY . /app`
	`RUN make /app`
	`RUN rm -r $HOME/.cache`
	`CMD python /app/app.py`
	This Dockerfile contains four commands. Commands that modify the filesystem create a layer. The `FROM` statement starts out by creating a layer from the `ubuntu:22.04` image. The `LABEL` command only modifies the image's metadata, and doesn't produce a new layer. The `COPY` command adds some files from your Docker client's current directory. The first `RUN` command builds your application using the `make` command, and writes the result to a new layer. The second `RUN` command removes a cache directory, and writes the result to a new layer. Finally, the `CMD` instruction specifies what command to run within the container, which only modifies the image's metadata, which doesn't produce an image layer.
	https://docs.docker.com/storage/storagedriver/

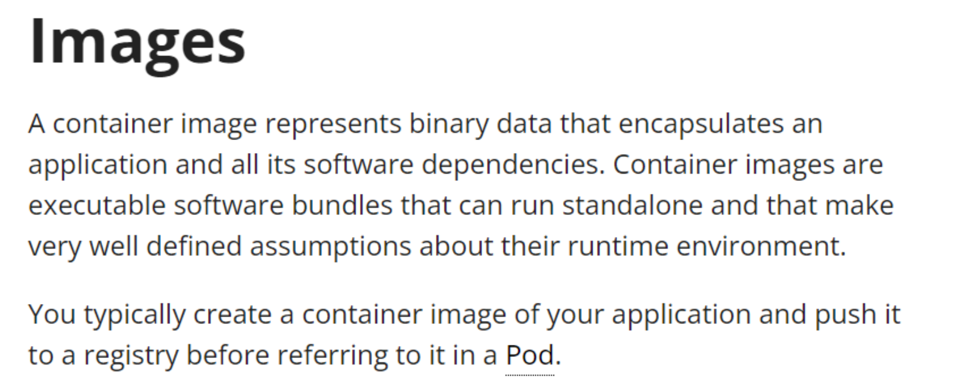
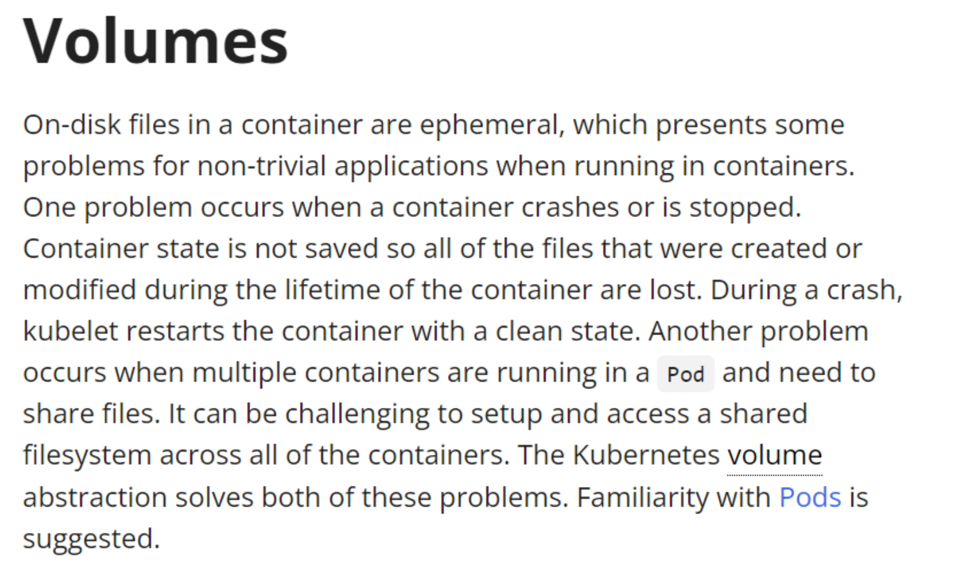
Each layer is only a set of differences from the layer before it. Note that both *adding*, and *removing* files will result in a new layer. In the example above, the `$HOME/.cache` directory is removed, but will still be available in the previous layer and add up to the image's total size. Refer to the Best practices for writing Dockerfiles and use multi-stage builds sections to learn how to optimize your Dockerfiles for efficient images.

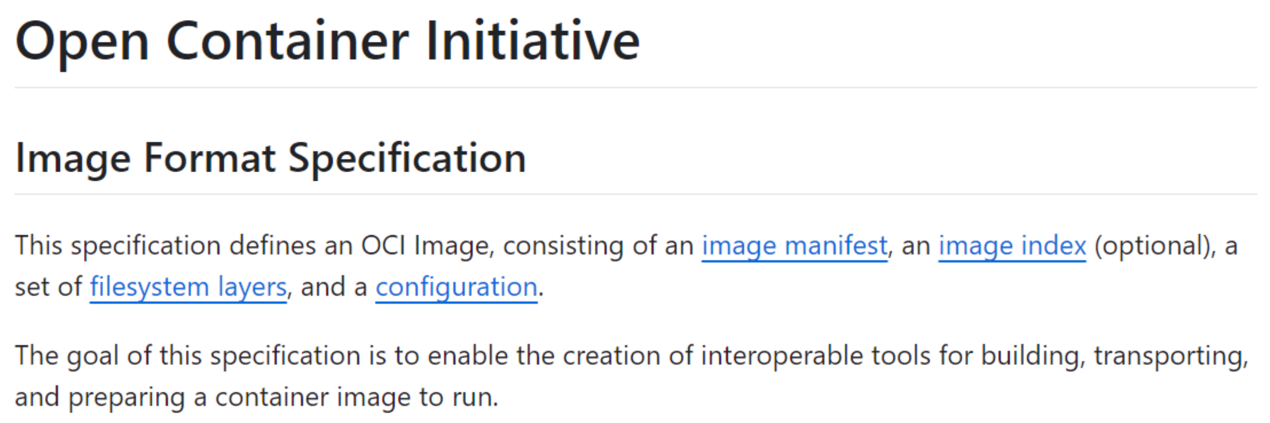
The layers are stacked on top of each other. When you create a new container, you add a new writable layer on top of the underlying layers. This layer is often called the "container layer". All changes made to the running container, such as writing new files, modifying existing files, and deleting files, are written to this thin writable container layer. The diagram below shows a container based on an `ubuntu:15.04` image.



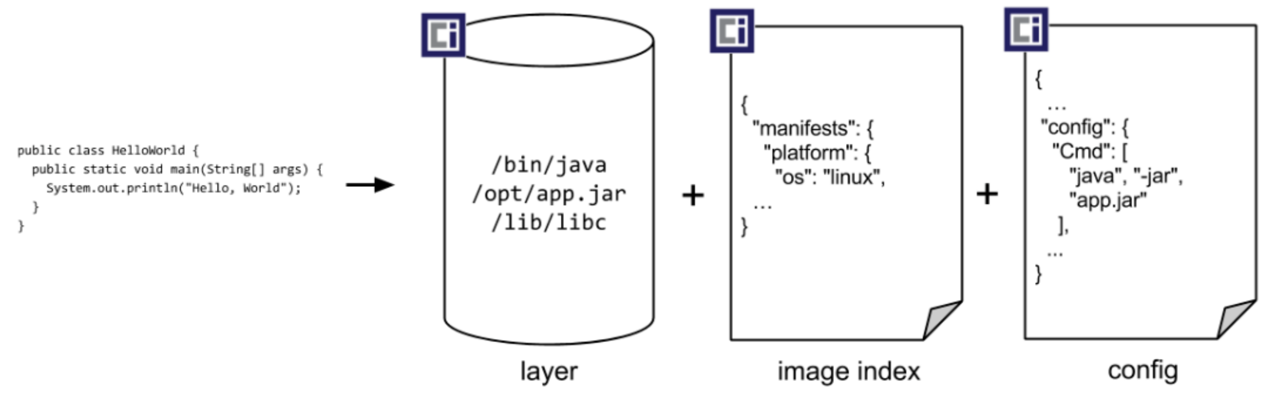
https://docs.docker.com/storage/storagedriver/

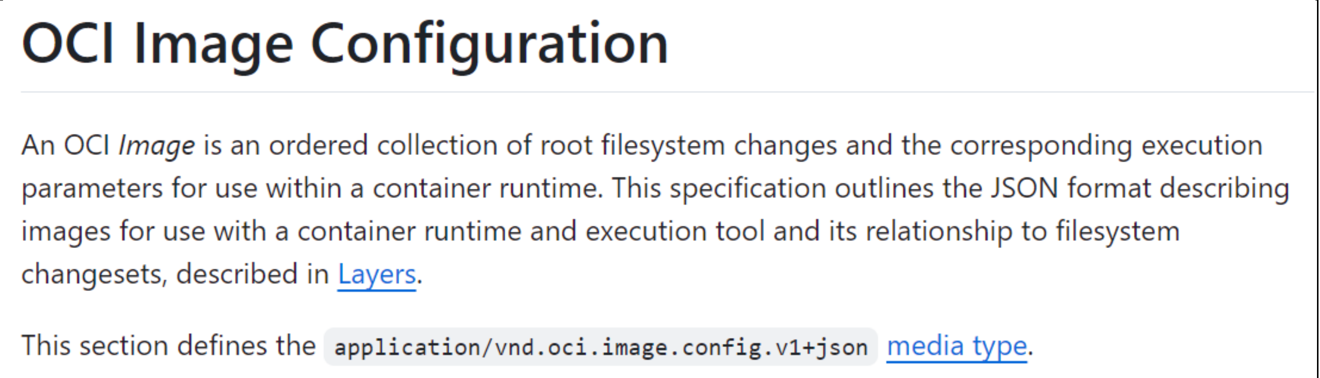
Claim 1	Accused Instrumentalities
	 **Volumes** Volumes are the preferred mechanism for persisting data generated by and used by Docker containers. While bind mounts are dependent on the directory structure and OS of the host machine, volumes are completely managed by Docker. Volumes have several advantages over bind mounts: https://kubernetes.io/docs/concepts/storage/volumes/  **Container environment** The Kubernetes Container environment provides several important resources to Containers: • A filesystem, which is a combination of an image and one or more volumes. • Information about the Container itself. • Information about other objects in the cluster. https://kubernetes.io/docs/concepts/containers/container-environment/

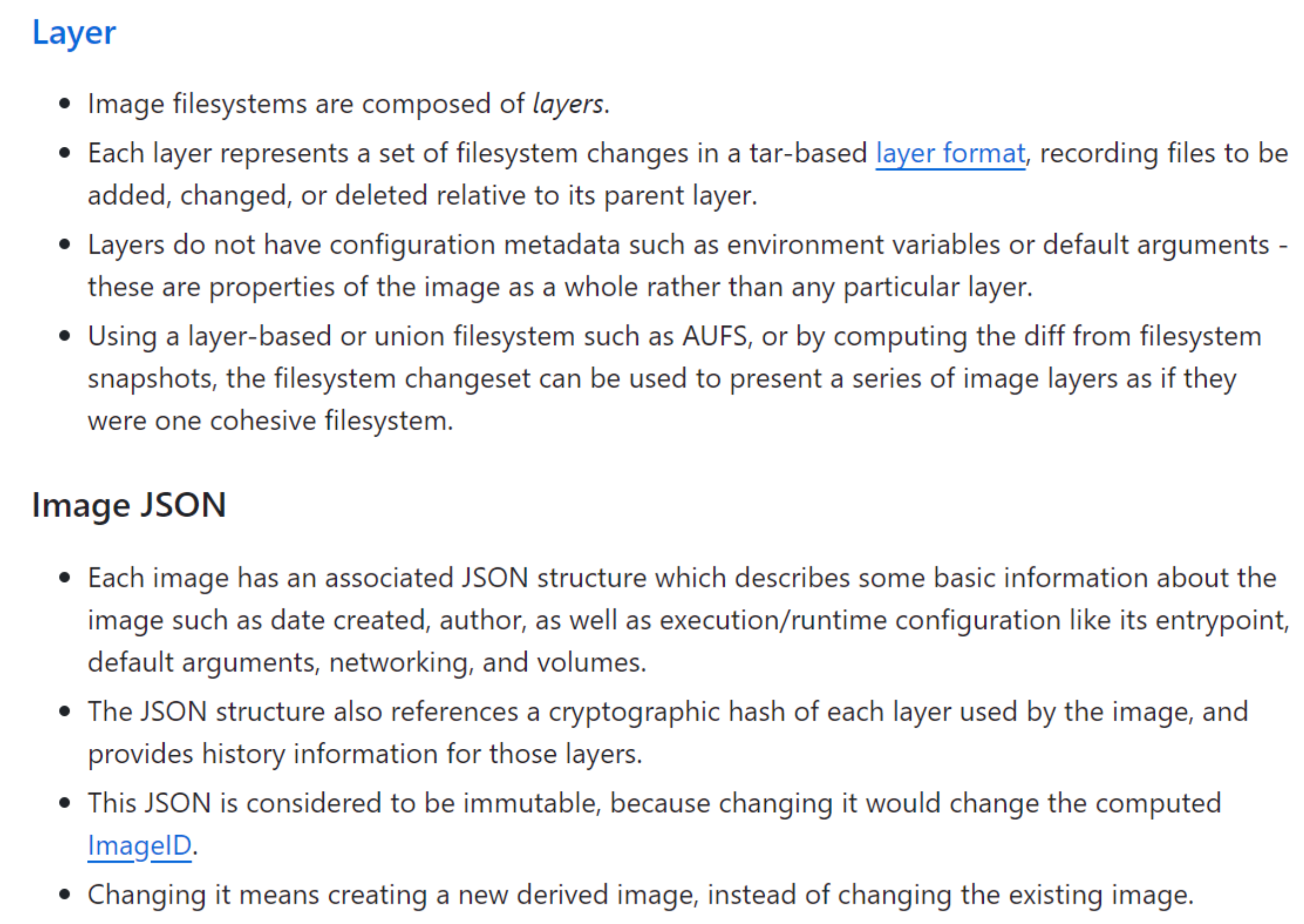
Claim 1	Accused Instrumentalities
	 **Images** A container image represents binary data that encapsulates an application and all its software dependencies. Container images are executable software bundles that can run standalone and that make very well defined assumptions about their runtime environment. You typically create a container image of your application and push it to a registry before referring to it in a Pod. https://kubernetes.io/docs/concepts/containers/images/  **Volumes** On-disk files in a container are ephemeral, which presents some problems for non-trivial applications when running in containers. One problem occurs when a container crashes or is stopped. Container state is not saved so all of the files that were created or modified during the lifetime of the container are lost. During a crash, kubelet restarts the container with a clean state. Another problem occurs when multiple containers are running in a `Pod` and need to share files. It can be challenging to setup and access a shared filesystem across all of the containers. The Kubernetes volume abstraction solves both of these problems. Familiarity with Pods is suggested. https://kubernetes.io/docs/concepts/storage/volumes/

Claim 1	Accused Instrumentalities
	 Open Container Initiative Image Format Specification This specification defines an OCI Image, consisting of an image manifest, an image index (optional), a set of filesystem layers, and a configuration. The goal of this specification is to enable the creation of interoperable tools for building, transporting, and preparing a container image to run. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/spec.md

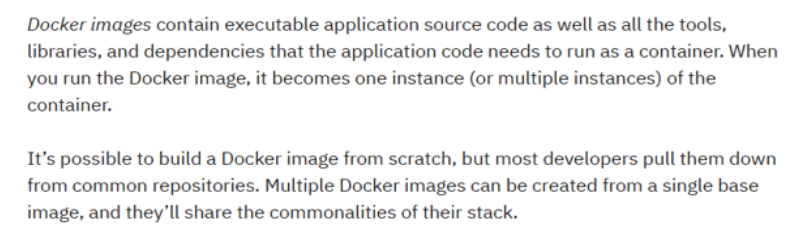
Claim 1	Accused Instrumentalities

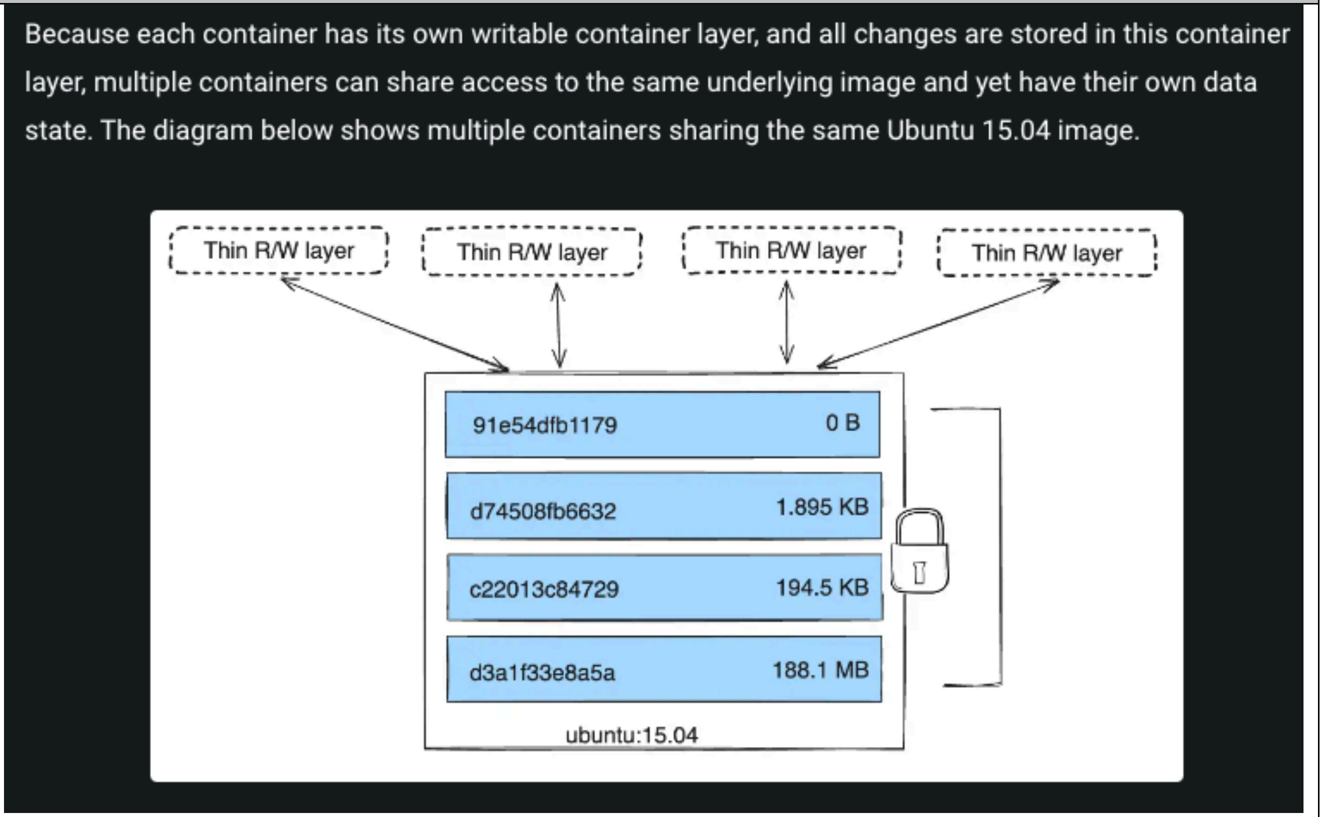
	Overview
	
	At a high level the image manifest contains metadata about the contents and dependencies of the image including the content-addressable identity of one or more filesystem layer changeset archives that will be unpacked to make up the final runnable filesystem. The image configuration includes information such as application arguments, environments, etc. The image index is a higher-level manifest which points to a list of manifests and descriptors. Typically, these manifests may provide different implementations of the image, possibly varying by platform or other attributes.
	
	
	`public class HelloWorld { public static void main(String[] args) { System.out.println("Hello, World"); } }`
	`{ "manifests": { "platform": { "os": "linux", ... }`
	`{ ... "config": { "Cmd": [ "java", "-jar", "app.jar" ], ... }`
	 /bin/java
	/opt/app.jar
	/lib/libc
	layer
	image index
	config
	
	https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/spec.md

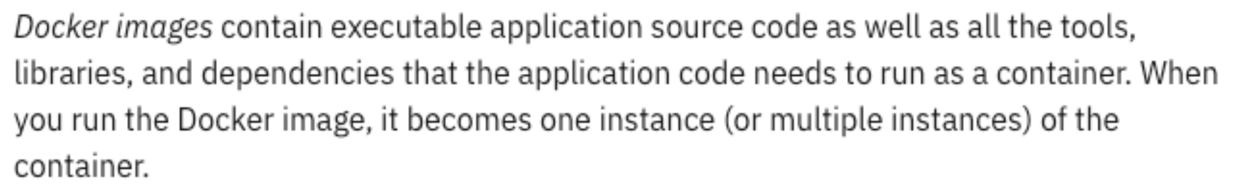
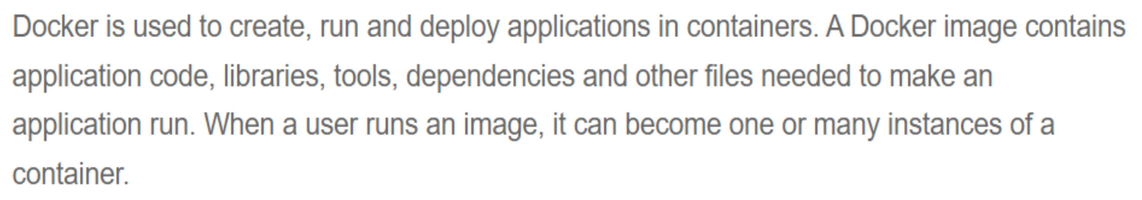
Claim 1	Accused Instrumentalities
	 OCI Image Configuration An OCI *Image* is an ordered collection of root filesystem changes and the corresponding execution parameters for use within a container runtime. This specification outlines the JSON format describing images for use with a container runtime and execution tool and its relationship to filesystem changesets, described in Layers. This section defines the `application/vnd.oci.image.config.v1+json` media type. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/config.md

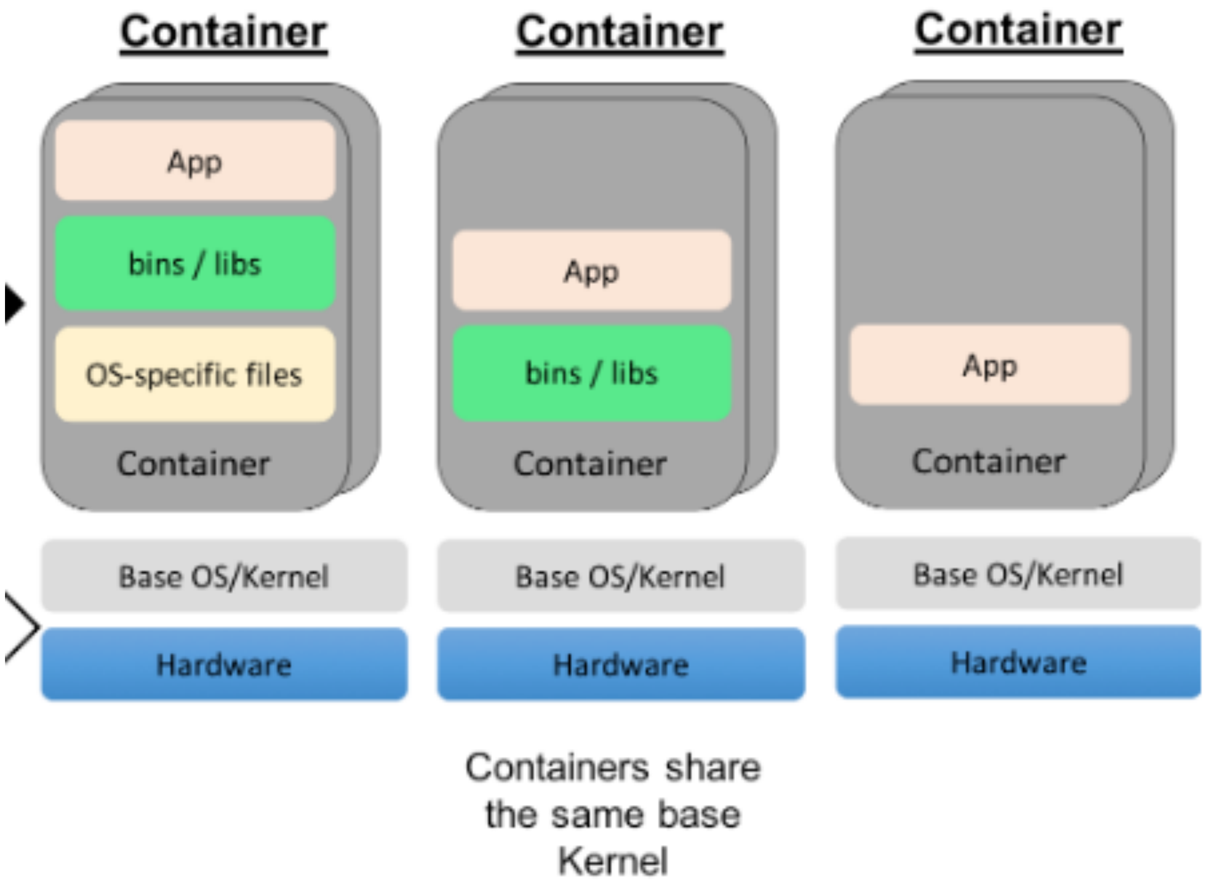
Claim 1	Accused Instrumentalities
	 **Layer** • Image filesystems are composed of *layers*. • Each layer represents a set of filesystem changes in a tar-based layer format, recording files to be added, changed, or deleted relative to its parent layer. • Layers do not have configuration metadata such as environment variables or default arguments - these are properties of the image as a whole rather than any particular layer. • Using a layer-based or union filesystem such as AUFS, or by computing the diff from filesystem snapshots, the filesystem changeset can be used to present a series of image layers as if they were one cohesive filesystem. **Image JSON** • Each image has an associated JSON structure which describes some basic information about the image such as date created, author, as well as execution/runtime configuration like its entrypoint, default arguments, networking, and volumes. • The JSON structure also references a cryptographic hash of each layer used by the image, and provides history information for those layers. • This JSON is considered to be immutable, because changing it would change the computed ImageID. • Changing it means creating a new derived image, instead of changing the existing image. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/config.md

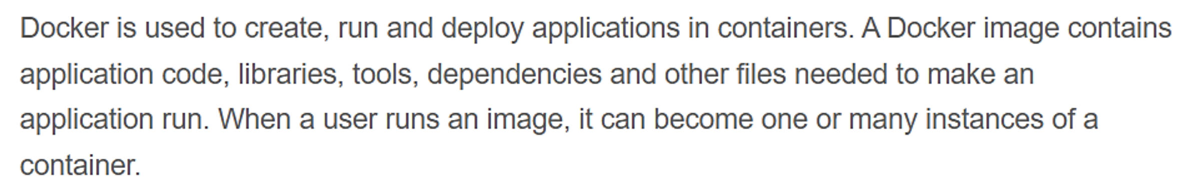
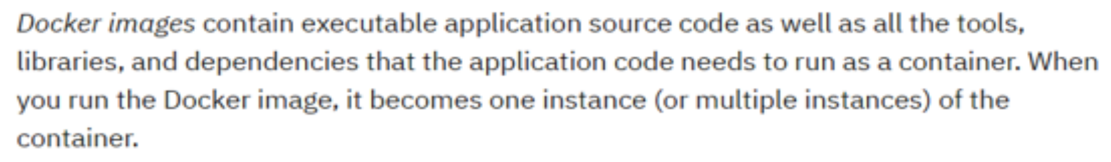
Claim 1	Accused Instrumentalities
[1e] ii) wherein an instance of a SLCSE provided to at least a first of the plurality of software applications from the shared library is run in a context of said at least first of the plurality of software applications without being shared with other of the plurality of software applications and where at least a second of the plurality of software applications running under the operating system have use of a unique instance of a corresponding critical system element for performing same function, and	In each Accused Instrumentality, an instance of a SLCSE provided to at least a first of the plurality of software applications from the shared library is run in a context of said at least first of the plurality of software applications without being shared with other of the plurality of software applications and where at least a second of the plurality of software applications running under the operating system have use of a unique instance of a corresponding critical system element for performing same function. When a Docker or Kubernetes image is used to create a container, it creates a separate and isolated instance of a runtime environment which is independent of other containers running on the same host. Each container has its own instance of base images and its own data. The containers run in isolation, ensuring that the SLCSEs stored in the shared library are accessible to the software applications running in their respective containers. The image includes essential system files, libraries, and dependencies required to run the software application within the container. The containers can share common dependencies and components using layered images. This means that multiple containers utilize the same base image to create an instance. When an instance of SLCSE is provided from the base image (i.e., from the shared library) to an individual container including application software, it operates in isolation and runs its own instance of the software application without sharing resources or critical system elements with other containers. This ensures that each container has its own isolated context. Docker or Kubernetes containers can share common dependencies and components using layered images. This means that multiple containers can utilize the same base image. Therefore, each container, containing the application software running under the operating system, utilizes a unique instance of the corresponding critical system element to execute the respective application software for performing a same or a different function. *See, e.g.*:

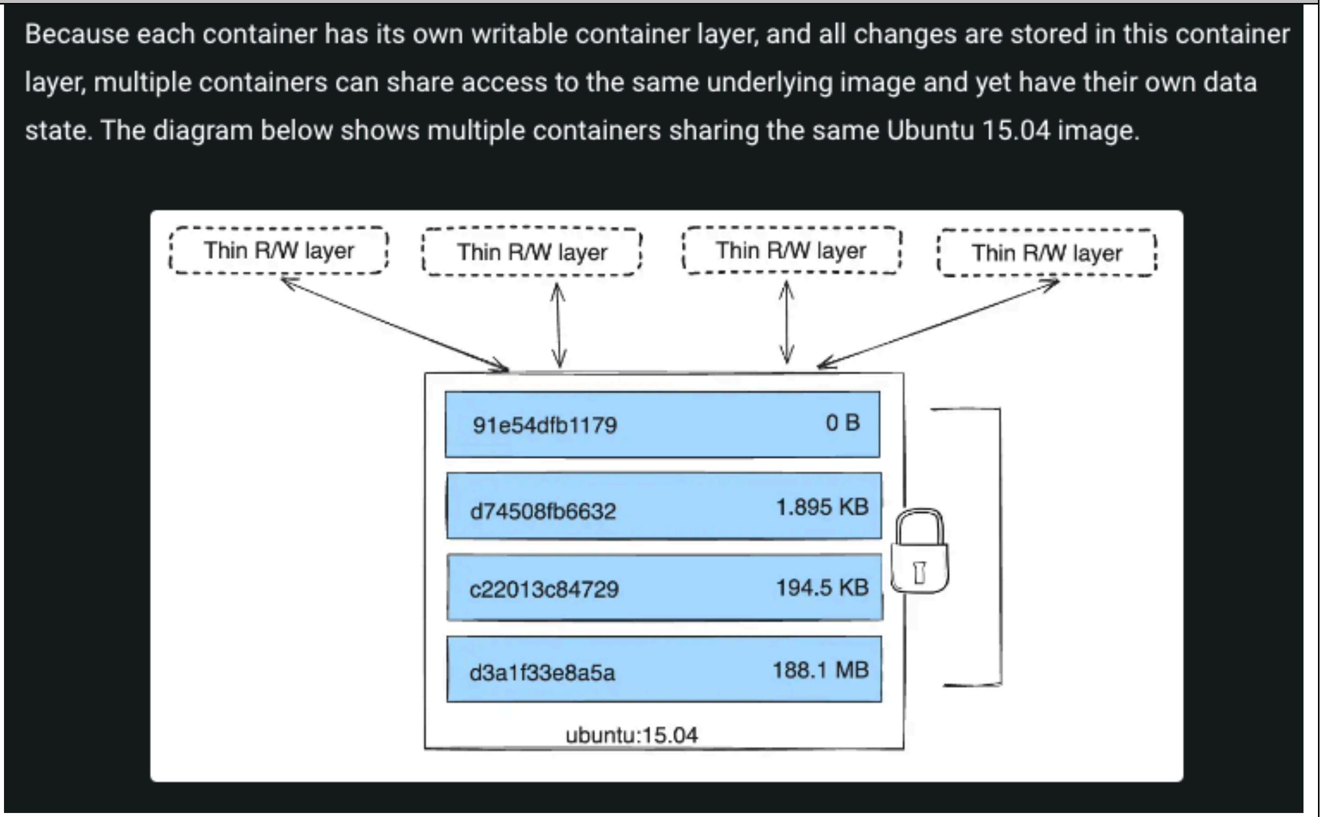
Claim 1	Accused Instrumentalities
	 *Docker images* contain executable application source code as well as all the tools, libraries, and dependencies that the application code needs to run as a container. When you run the Docker image, it becomes one instance (or multiple instances) of the container. It's possible to build a Docker image from scratch, but most developers pull them down from common repositories. Multiple Docker images can be created from a single base image, and they'll share the commonalities of their stack. https://www.ibm.com/topics/docker

Claim 1	Accused Instrumentalities
	 Because each container has its own writable container layer, and all changes are stored in this container layer, multiple containers can share access to the same underlying image and yet have their own data state. The diagram below shows multiple containers sharing the same Ubuntu 15.04 image. Thin R/W layer Thin R/W layer Thin R/W layer Thin R/W layer 91e54dfb1179 0 B d74508fb6632 1.895 KB c22013c84729 194.5 KB d3a1f33e8a5a 188.1 MB ubuntu:15.04 https://docs.docker.com/storage/storagedriver/

Claim 1	Accused Instrumentalities
	 *Docker images* contain executable application source code as well as all the tools, libraries, and dependencies that the application code needs to run as a container. When you run the Docker image, it becomes one instance (or multiple instances) of the container. https://www.ibm.com/topics/docker  Docker is used to create, run and deploy applications in containers. A Docker image contains application code, libraries, tools, dependencies and other files needed to make an application run. When a user runs an image, it can become one or many instances of a container. https://www.techtarget.com/searchitoperations/definition/Docker-image

Claim 1	Accused Instrumentalities
	 Container Container Container App bins / libs OS-specific files Container App bins / libs Container App Container Base OS/Kernel Base OS/Kernel Base OS/Kernel Hardware Hardware Hardware Containers share the same base Kernel https://ibm.github.io/kube101/

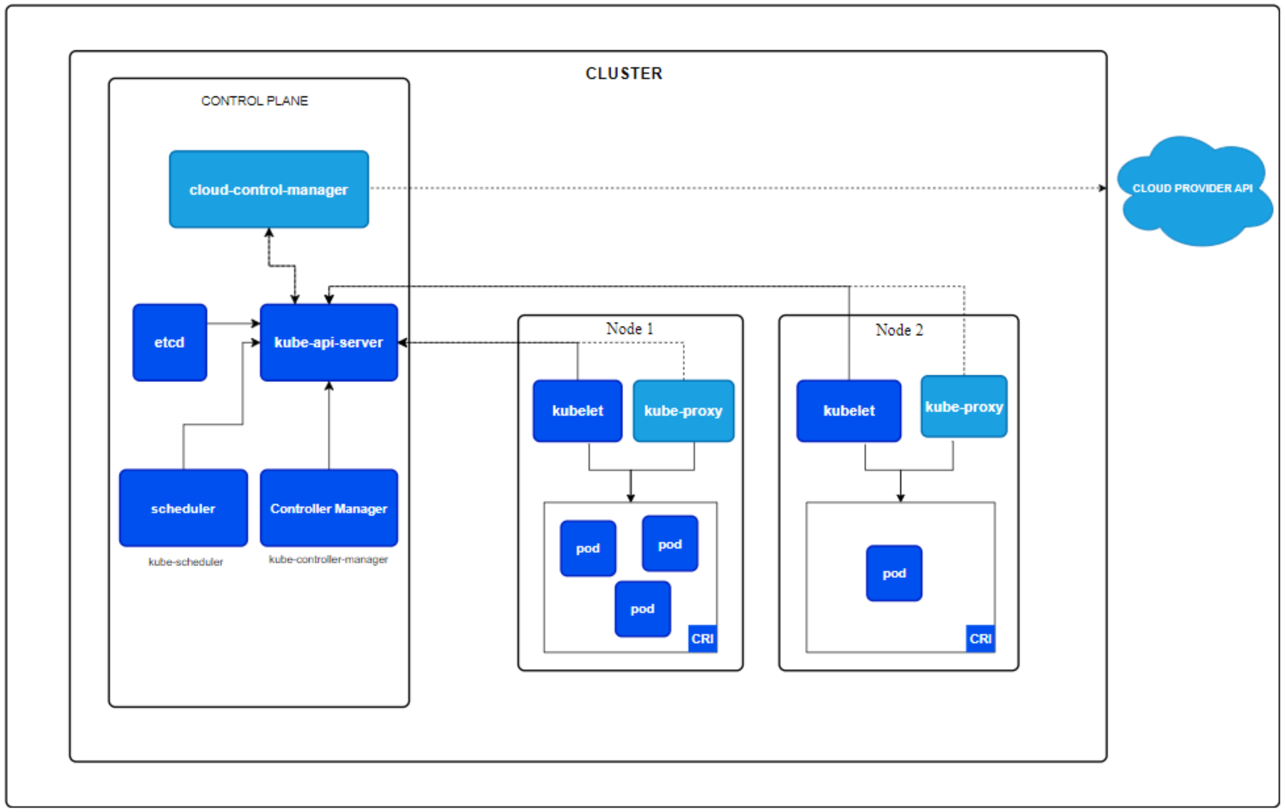
Claim 1	Accused Instrumentalities
[1f] iii) wherein a SLCSE related to a predetermined function is provided to the first of the plurality of software applications for running a first instance of the SLCSE, and wherein a SLCSE for performing a same function is provided to the second of the plurality of software applications for running a second instance of the SLCSE simultaneously.	In each Accused Instrumentality, a SLCSE related to a predetermined function is provided to the first of the plurality of software applications for running a first instance of the SLCSE, and wherein a SLCSE for performing a same function is provided to the second of the plurality of software applications for running a second instance of the SLCSE simultaneously. For example, in Docker or Kubernetes containers, each container operates independently, and a base image includes essential system files, libraries, and dependencies (i.e., SLCSEs) required to run the software application within the container. Based on information and belief, each element, such as system files, libraries, and dependencies (i.e., SLCSE) is associated with an execution of a predetermined function related to the application. When an image is used to create a container in the Accused Instrumentality, an instance of the SLCSE is provided to a software application. Therefore, different instances of the SLCSE are provided to different applications for performing either a same or a different function, simultaneously. *See, e.g.*:  Docker is used to create, run and deploy applications in containers. A Docker image contains application code, libraries, tools, dependencies and other files needed to make an application run. When a user runs an image, it can become one or many instances of a container. https://www.techtarget.com/searchitoperations/definition/Docker-image  *Docker images* contain executable application source code as well as all the tools, libraries, and dependencies that the application code needs to run as a container. When you run the Docker image, it becomes one instance (or multiple instances) of the container. https://www.ibm.com/topics/docker

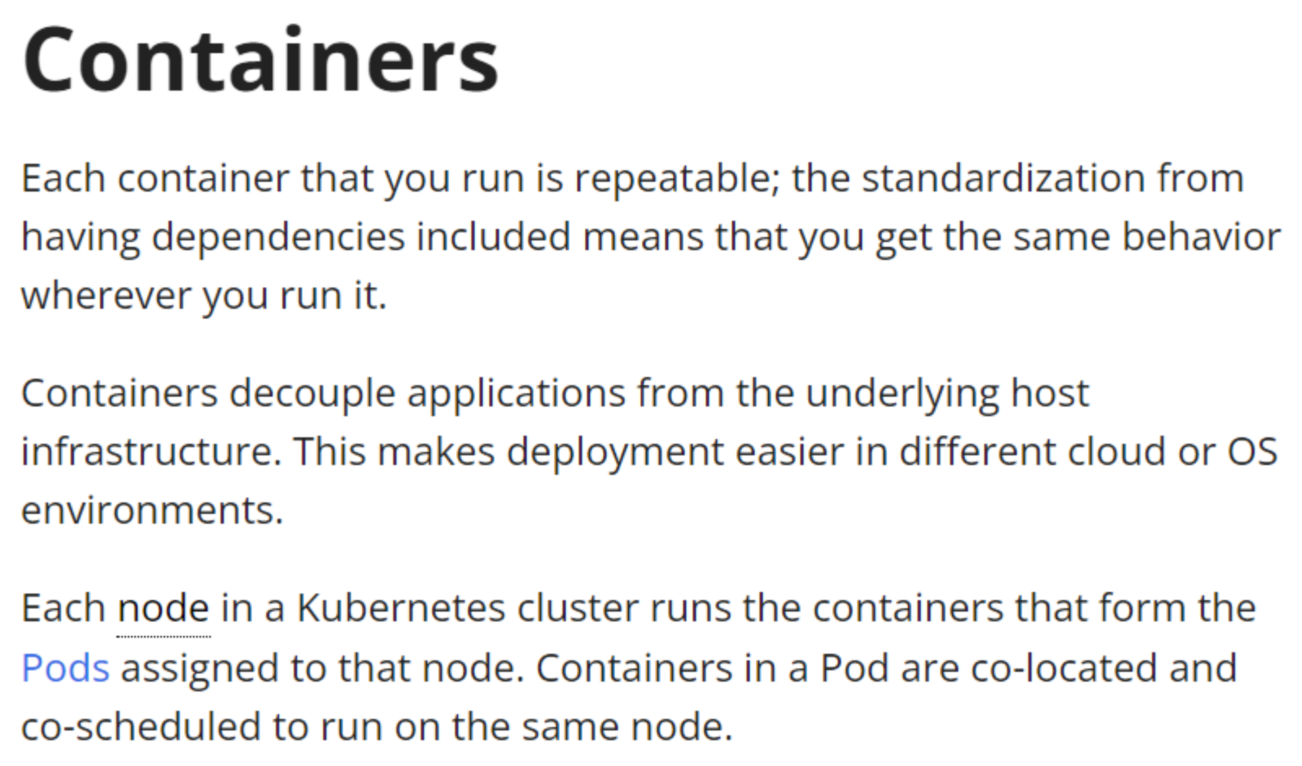
Claim 1	Accused Instrumentalities
	 Because each container has its own writable container layer, and all changes are stored in this container layer, multiple containers can share access to the same underlying image and yet have their own data state. The diagram below shows multiple containers sharing the same Ubuntu 15.04 image. Thin R/W layer Thin R/W layer Thin R/W layer Thin R/W layer 91e54dfb1179 0 B d74508fb6632 1.895 KB c22013c84729 194.5 KB d3a1f33e8a5a 188.1 MB ubuntu:15.04 https://docs.docker.com/storage/storagedriver/ A container is a runnable instance of an image. You can create, start, stop, move, or delete a container using the Docker API or CLI. You can connect a container to one or more networks, attach storage to it, or even create a new image based on its current state.

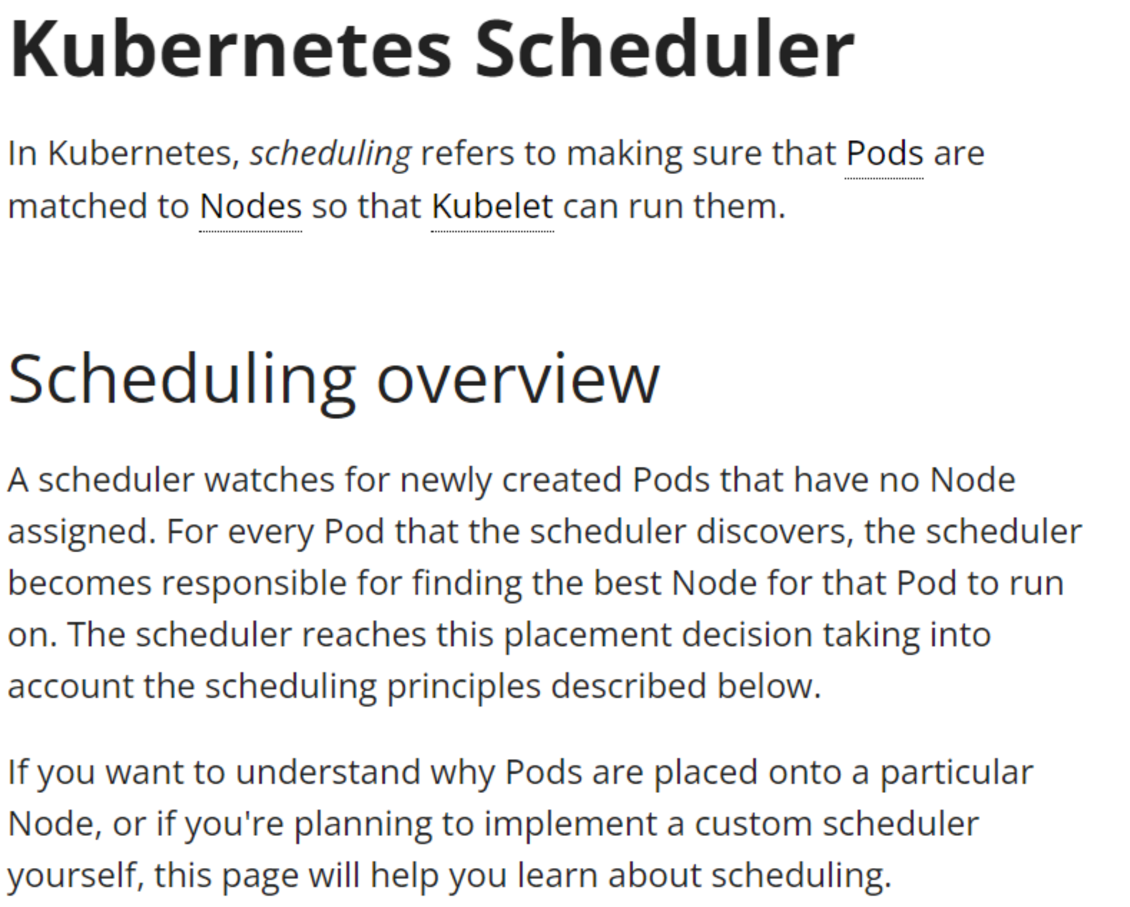
Claim 1	Accused Instrumentalities
	https://docs.docker.com/get-started/overview/

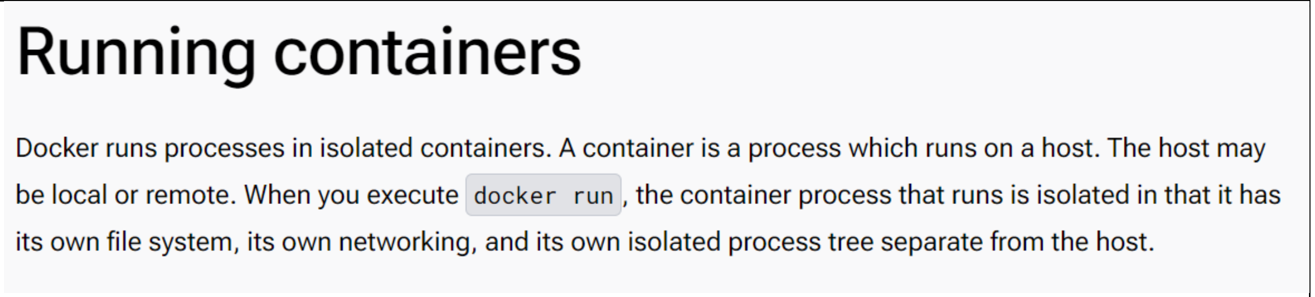
Claim 2

Claim 2	**Accused Instrumentalities**
2. A computing system as defined in claim 1, wherein in operation, multiple instances of an SLCSE stored in the shared library run simultaneously within the operating system.	Each Accused Instrumentality comprises or constitutes a computing system as defined in claim 1, wherein in operation, multiple instances of an SLCSE stored in the shared library run simultaneously within the operating system. For example, an individual host/node runs multiple containers and/or pods simultaneously, each of which has an instance of an SLCSE. *See, e.g.*:

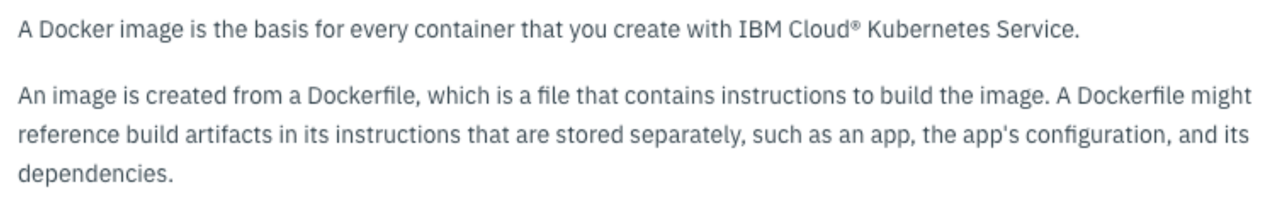
Claim 2	Accused Instrumentalities
	 CLUSTER CONTROL PLANE cloud-control-manager etcd kube-api-server scheduler kube-scheduler Controller Manager kube-controller-manager Node 1 kubelet kube-proxy pod pod pod CRI Node 2 kubelet kube-proxy pod CRI CLOUD PROVIDER API Kubernetes cluster architecture https://kubernetes.io/docs/concepts/architecture/

Claim 2	Accused Instrumentalities
	 **Containers** Each container that you run is repeatable; the standardization from having dependencies included means that you get the same behavior wherever you run it. Containers decouple applications from the underlying host infrastructure. This makes deployment easier in different cloud or OS environments. Each node in a Kubernetes cluster runs the containers that form the Pods assigned to that node. Containers in a Pod are co-located and co-scheduled to run on the same node. https://kubernetes.io/docs/concepts/containers/

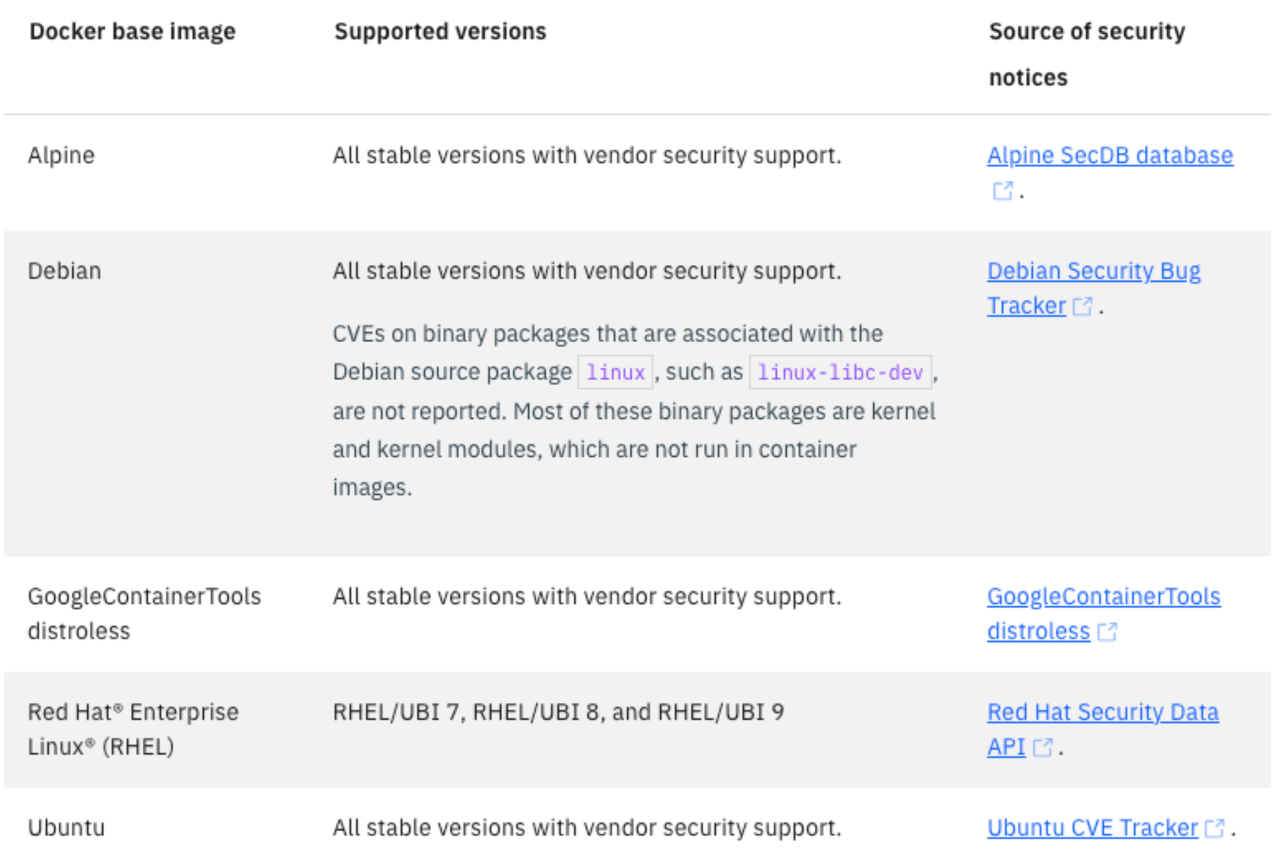
Claim 2	Accused Instrumentalities
	 **Kubernetes Scheduler** In Kubernetes, *scheduling* refers to making sure that Pods are matched to Nodes so that Kubelet can run them. **Scheduling overview** A scheduler watches for newly created Pods that have no Node assigned. For every Pod that the scheduler discovers, the scheduler becomes responsible for finding the best Node for that Pod to run on. The scheduler reaches this placement decision taking into account the scheduling principles described below. If you want to understand why Pods are placed onto a particular Node, or if you're planning to implement a custom scheduler yourself, this page will help you learn about scheduling. https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/

Claim 2	Accused Instrumentalities
	 **Running containers** Docker runs processes in isolated containers. A container is a process which runs on a host. The host may be local or remote. When you execute `docker run`, the container process that runs is isolated in that it has its own file system, its own networking, and its own isolated process tree separate from the host. https://docs.docker.com/engine/reference/run/

Claim 3

Claim 3	Accused Instrumentalities
3. A computing system according to claim 1 wherein OSCSEs corresponding to and capable of performing the same function as SLCSEs remain in the operating system kernel.	Each Accused Instrumentality comprises or constitutes a computing system according to claim 1 wherein OSCSEs corresponding to and capable of performing the same function as SLCSEs remain in the operating system kernel. For example, both Docker and Kubernetes systems preserve the host kernel substantially unchanged; therefore the OSCSEs corresponding to the SLCSEs remain in the operating system kernel. *See, e.g.*:  A Docker image is the basis for every container that you create with IBM Cloud® Kubernetes Service. An image is created from a Dockerfile, which is a file that contains instructions to build the image. A Dockerfile might reference build artifacts in its instructions that are stored separately, such as an app, the app's configuration, and its dependencies. https://cloud.ibm.com/docs/containers?topic=containers-images

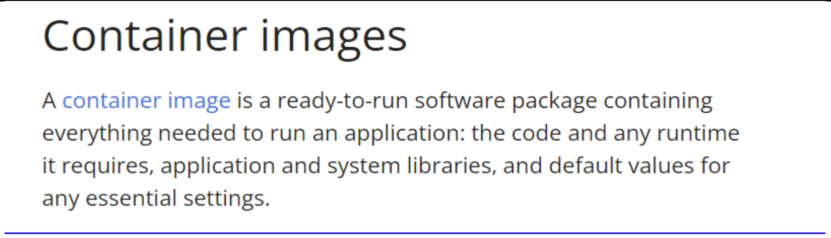
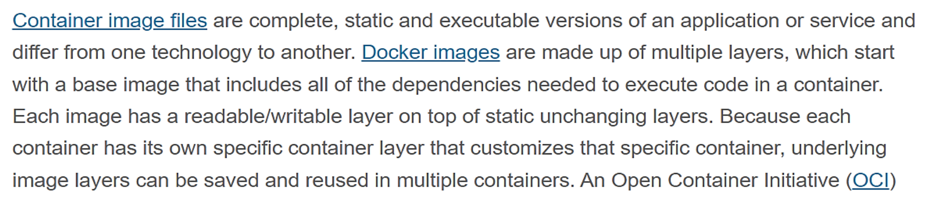
<table>
<tr><th>Claim 3</th><th>Accused Instrumentalities</th></tr>
<tr><td></td><td>

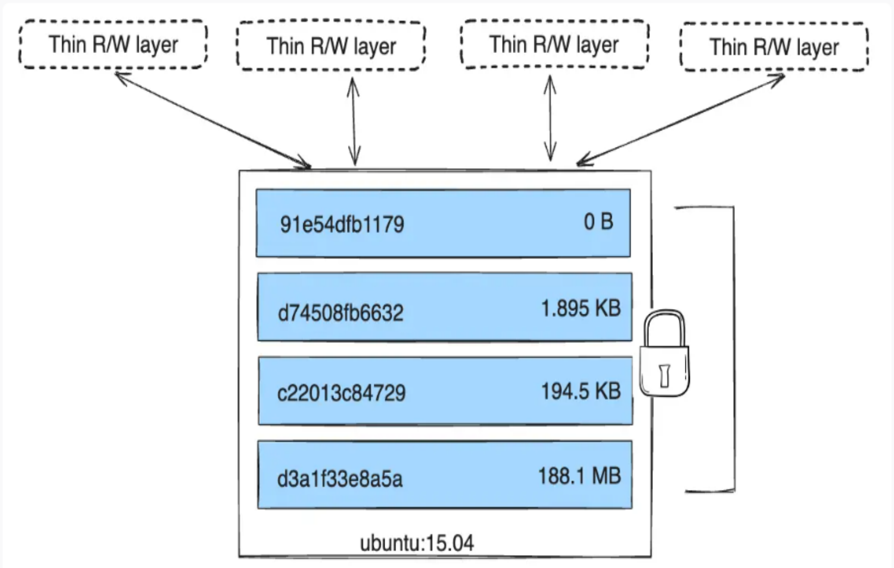


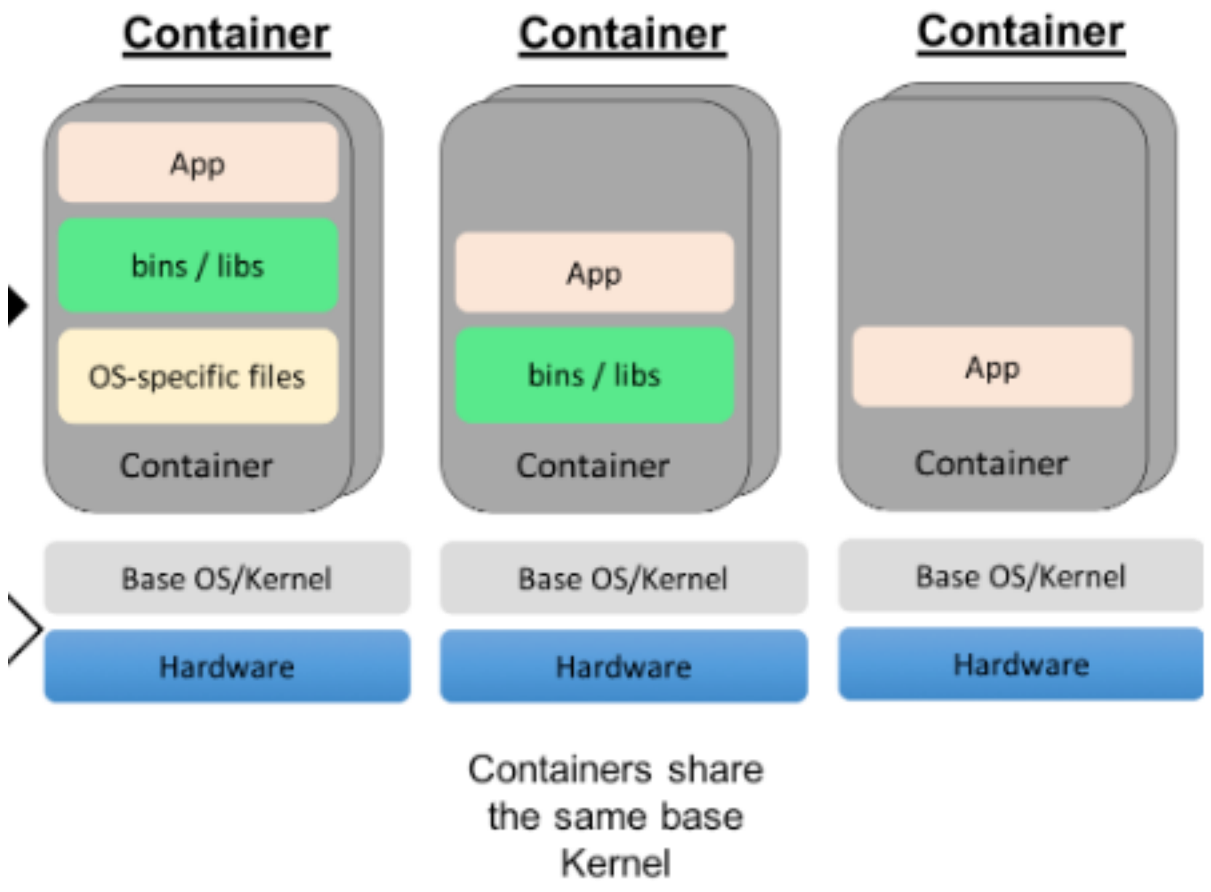
Docker base image	Supported versions	Source of security notices
Alpine	All stable versions with vendor security support.	Alpine SecDB database.
Debian	All stable versions with vendor security support. CVEs on binary packages that are associated with the Debian source package `linux`, such as `linux-libc-dev`, are not reported. Most of these binary packages are kernel and kernel modules, which are not run in container images.	Debian Security Bug Tracker.
GoogleContainerTools distroless	All stable versions with vendor security support.	GoogleContainerTools distroless
Red Hat® Enterprise Linux® (RHEL)	RHEL/UBI 7, RHEL/UBI 8, and RHEL/UBI 9	Red Hat Security Data API.
Ubuntu	All stable versions with vendor security support.	Ubuntu CVE Tracker.

https://cloud.ibm.com/docs/Registry?topic=Registry-va_index&interface=ui

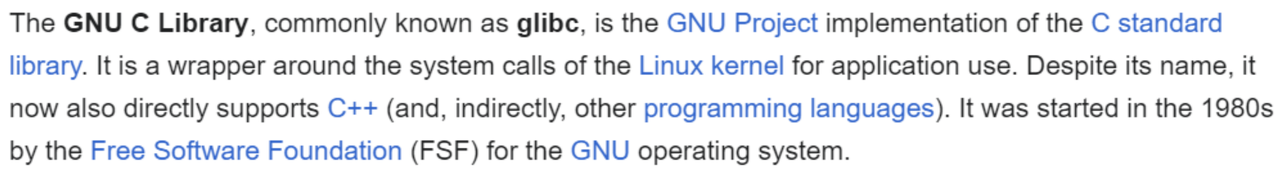
</td></tr>
</table>

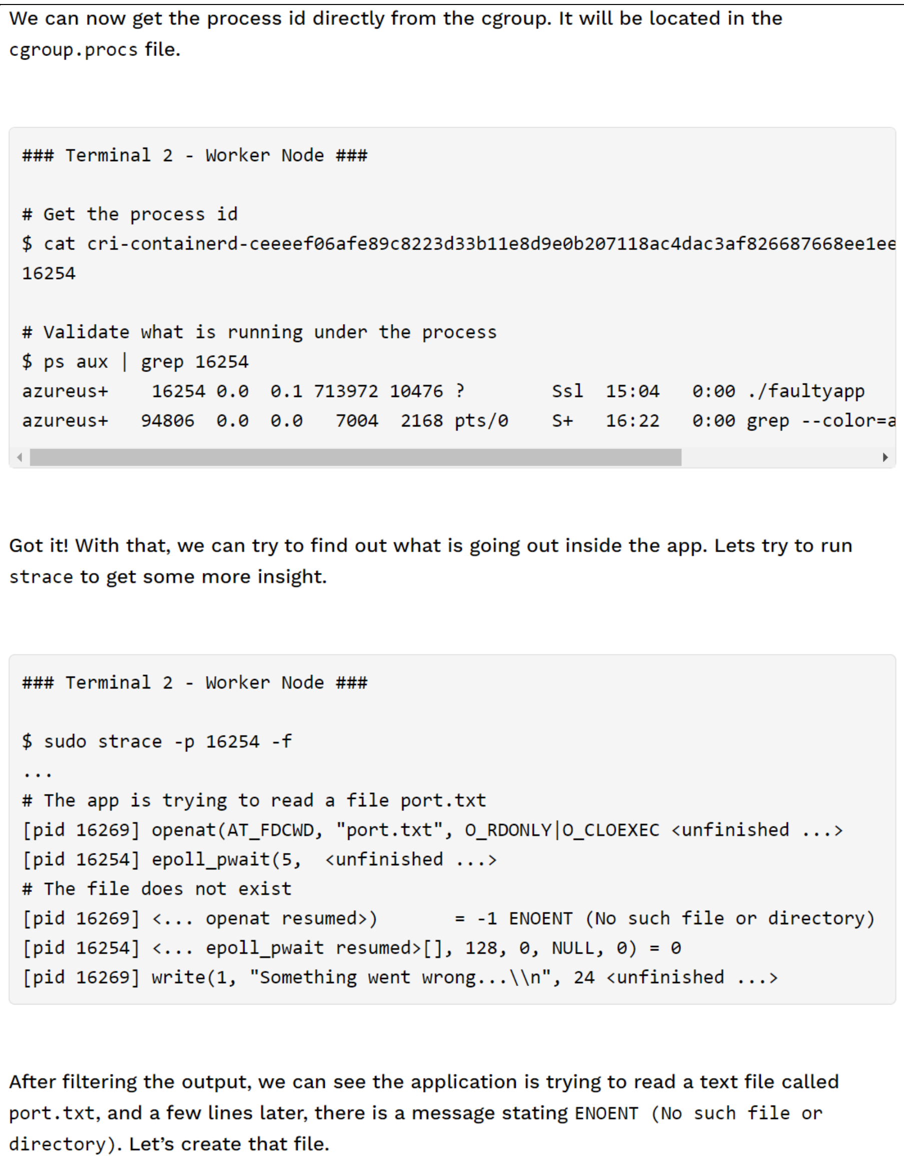
Claim 3	Accused Instrumentalities
	 Container images A container image is a ready-to-run software package containing everything needed to run an application: the code and any runtime it requires, application and system libraries, and default values for any essential settings. https://kubernetes.io/docs/concepts/containers/  Container image files are complete, static and executable versions of an application or service and differ from one technology to another. Docker images are made up of multiple layers, which start with a base image that includes all of the dependencies needed to execute code in a container. Each image has a readable/writable layer on top of static unchanging layers. Because each container has its own specific container layer that customizes that specific container, underlying image layers can be saved and reused in multiple containers. An Open Container Initiative (OCI) https://www.techtarget.com/searchitoperations/definition/container-containerization-or-container-based-virtualization

Claim 3	Accused Instrumentalities
	Because each container has its own writable container layer, and all changes are stored in this container layer, multiple containers can share access to the same underlying image and yet have their own data state. The diagram below shows multiple containers sharing the same Ubuntu 15.04 image.  Thin R/W layer Thin R/W layer Thin R/W layer Thin R/W layer 91e54dfb1179 0 B d74508fb6632 1.895 KB c22013c84729 194.5 KB d3a1f33e8a5a 188.1 MB ubuntu:15.04 https://docs.docker.com/storage/storagedriver/

Claim 3	Accused Instrumentalities
	 Container Container Container App bins / libs OS-specific files Container App bins / libs Container App Container Base OS/Kernel Base OS/Kernel Base OS/Kernel Hardware Hardware Hardware Containers share the same base Kernel https://ibm.github.io/kube101/

Claim 4

Claim 4	Accused Instrumentalities
4. A computing system according to claim 1 wherein the one or more SLCSEs provided to one of the plurality of software applications having exclusive use thereof, use system calls to access services in the operating system kernel.	Each Accused Instrumentality comprises or constitutes a computing system according to claim 1 wherein the one or more SLCSEs provided to one of the plurality of software applications having exclusive use thereof, use system calls to access services in the operating system kernel. For example, the SLCSEs in a container use system calls to access services in the operating system kernel. For example, the glibc library (or other similar library) in the container uses system calls to interface with the host Linux kernel. In general, system calls can be observed using a tool such as strace. *See, e.g.*:  The **GNU C Library**, commonly known as **glibc**, is the GNU Project implementation of the C standard library. It is a wrapper around the system calls of the Linux kernel for application use. Despite its name, it now also directly supports C++ (and, indirectly, other programming languages). It was started in the 1980s by the Free Software Foundation (FSF) for the GNU operating system. https://en.wikipedia.org/wiki/Glibc



We can now get the process id directly from the cgroup. It will be located in the `cgroup.procs` file.

```
### Terminal 2 - Worker Node ###

# Get the process id
$ cat cri-containerd-ceeeef06afe89c8223d33b11e8d9e0b207118ac4dac3af826687668ee1ee
16254

# Validate what is running under the process
$ ps aux | grep 16254
azureus+    16254 0.0  0.1 713972 10476 ?       Ssl  15:04   0:00 ./faultyapp
azureus+   94806  0.0  0.0   7004  2168 pts/0    S+   16:22   0:00 grep --color=a
```

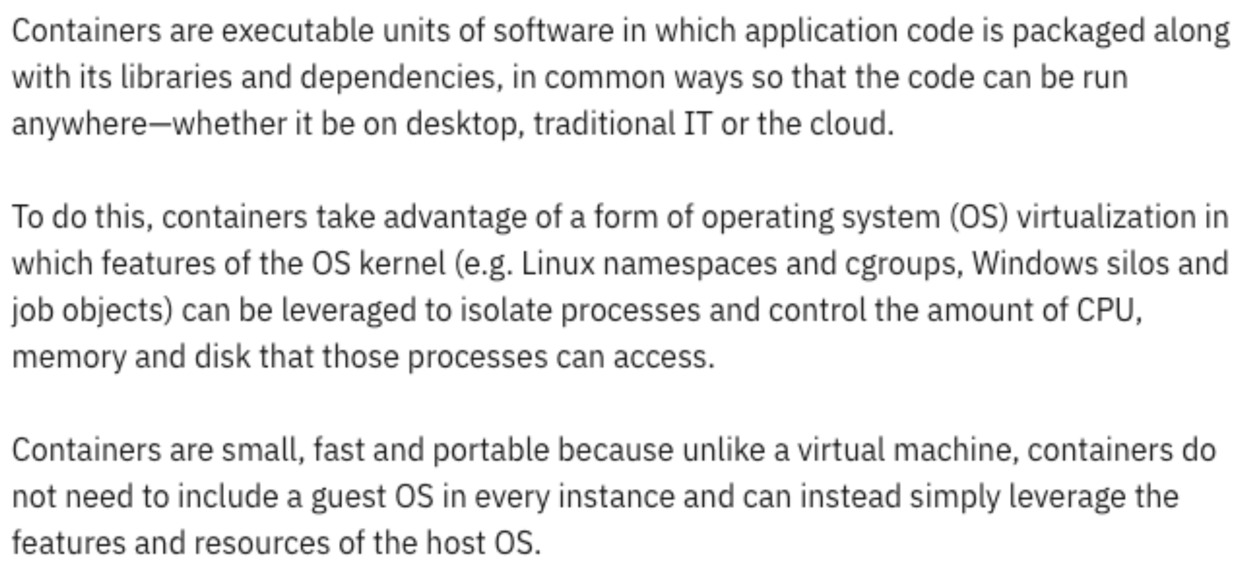
Got it! With that, we can try to find out what is going out inside the app. Lets try to run `strace` to get some more insight.

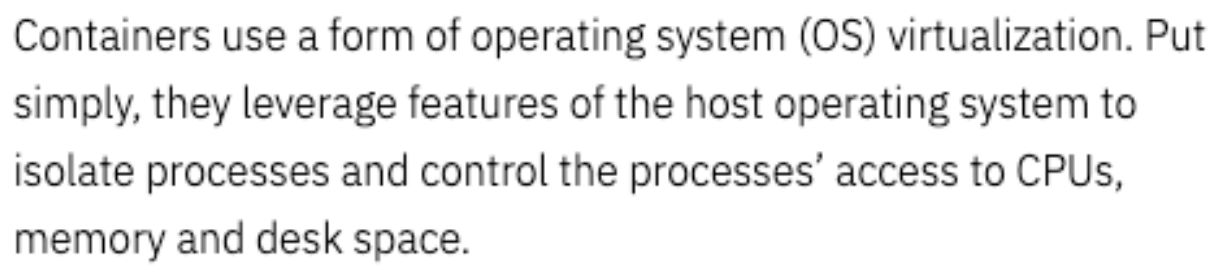
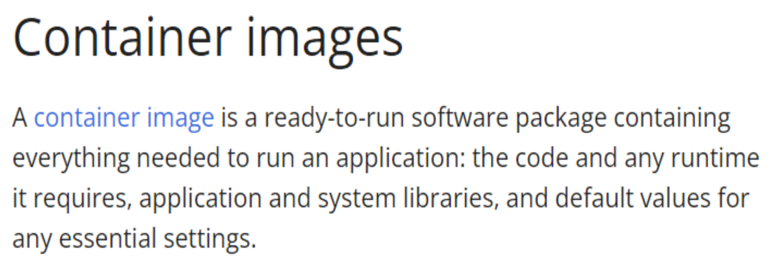
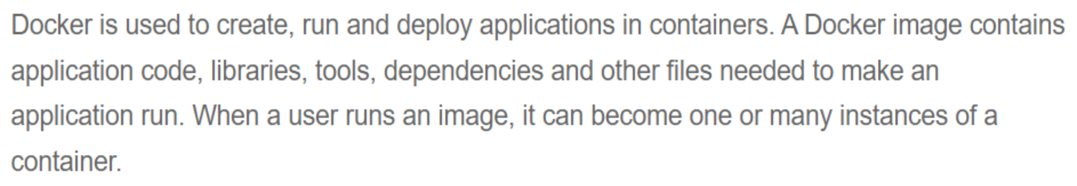
```
### Terminal 2 - Worker Node ###

$ sudo strace -p 16254 -f
...
# The app is trying to read a file port.txt
[pid 16269] openat(AT_FDCWD, "port.txt", O_RDONLY|O_CLOEXEC <unfinished ...>
[pid 16254] epoll_pwait(5,  <unfinished ...>
# The file does not exist
[pid 16269] <... openat resumed>)       = -1 ENOENT (No such file or directory)
[pid 16254] <... epoll_pwait resumed>[], 128, 0, NULL, 0) = 0
[pid 16269] write(1, "Something went wrong...\\n", 24 <unfinished ...>
```

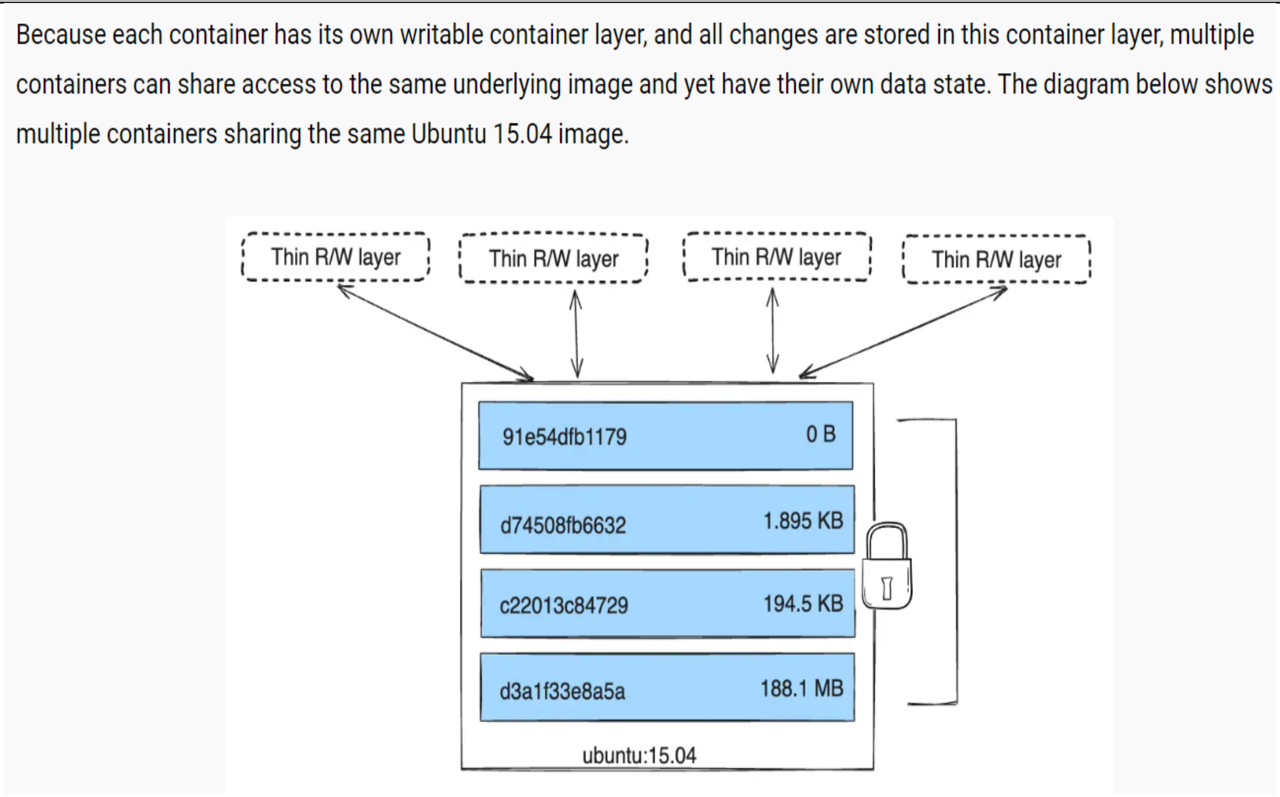
After filtering the output, we can see the application is trying to read a text file called `port.txt`, and a few lines later, there is a message stating `ENOENT (No such file or directory)`. Let's create that file.

https://www.berops.com/blog/a-different-method-to-debug-kubernetes-pods

Claim 18

Claim 18	Accused Instrumentalities
18. A computer system as defined in claim 2 wherein SLCSEs are not copies of OSCSEs.	Each Accused Instrumentality comprises or constitutes a computer system as defined in claim 2 wherein SLCSEs are not copies of OSCSEs. For example, in a typical case the SLCSEs come from a Linux distribution independent of the host operating system, and thus are not identical to the OSCSEs. For another example, the SLCSEs are provided to the computer system through a separate process than the process by which the OSCSEs are provided to the computer system, and thus are not copied from the OSCSEs. *See, e.g.*:  Containers are executable units of software in which application code is packaged along with its libraries and dependencies, in common ways so that the code can be run anywhere—whether it be on desktop, traditional IT or the cloud. To do this, containers take advantage of a form of operating system (OS) virtualization in which features of the OS kernel (e.g. Linux namespaces and cgroups, Windows silos and job objects) can be leveraged to isolate processes and control the amount of CPU, memory and disk that those processes can access. Containers are small, fast and portable because unlike a virtual machine, containers do not need to include a guest OS in every instance and can instead simply leverage the features and resources of the host OS. https://www.ibm.com/topics/containers

Claim 18	Accused Instrumentalities
	 Containers use a form of operating system (OS) virtualization. Put simply, they leverage features of the host operating system to isolate processes and control the processes' access to CPUs, memory and desk space. https://www.ibm.com/blog/containers-vs-vms/  Container images A container image is a ready-to-run software package containing everything needed to run an application: the code and any runtime it requires, application and system libraries, and default values for any essential settings. https://kubernetes.io/docs/concepts/containers/  Docker is used to create, run and deploy applications in containers. A Docker image contains application code, libraries, tools, dependencies and other files needed to make an application run. When a user runs an image, it can become one or many instances of a container. ttps://www.techtarget.com/searchitoperations/definition/Docker-image

Claim 18	Accused Instrumentalities
	 Because each container has its own writable container layer, and all changes are stored in this container layer, multiple containers can share access to the same underlying image and yet have their own data state. The diagram below shows multiple containers sharing the same Ubuntu 15.04 image. Thin R/W layer Thin R/W layer Thin R/W layer Thin R/W layer 91e54dfb1179 0 B d74508fb6632 1.895 KB c22013c84729 194.5 KB d3a1f33e8a5a 188.1 MB ubuntu:15.04 https://docs.docker.com/storage/storagedriver/